DTS202TC Foundation of Parallel Computing

Lecture 1

Hong-Bin Liu

Xi'an Jiaotong Liverpool University

September 6, 2021

About me

- Hong-Bin Liu is a new assistant professor at School of AI and Advanced Computing (SAAC). Before joining XJTLU, he obtained his PhD from James Cook University and Master of Computer Science from RMIT, Australia. And he worked as a full-stack developer for 5 years both in China and Australia. His research interests include Artificial Intelligence, Spatio-Temporal Reasoning, and Computer Vision etc.
- Email: hongbin.liu@xjtlu.edu.cn
- Office: SC540D

Module Overview

- Prerequisites: DTS102TC and MTH007
- Learning outcomes:
  - Identify serial and parallel algorithms.
  - Devise and implement parallel algorithms.
  - Identify and solve a computation problem with parallel algorithm.

Module Content

- Week 1 (Sep. 6-12): Introduction and C review.
- Week 2 (Sep. 13-19): Distributed-Memory Programming with MPI
- Week 3 (Sep. 20-26): Share-Memory Programming with Pthreads
- Week 4 (Sep. 27 - Oct. 3) : Shared-Memory Programming with OpenMP
- Week 5 (Oct. 11-17): Intro to CUDA (Course provided by Nvidia)

Module Delivery Plan

- Week 1-4: Online delivery
  - Recorded lectures and tutorials (no live sessions).
  - Lab Q&A live sessions on every Friday.
- Week 5 Face-to-face delivery as planned

Assignments

- Group Assignment 1, 20%, due October 10, 23:59pm (Week 4)
- Group Assignment 2, 30%, due October 24, 23:59pm (Week 6)
- Lab Report, 50%, due October 31, 23:59pm (Week 7)

Text Book and readings

- Peter Pacheco, An Introduction to Parallel Programming, Morgan-Kaufmann, 2011.
- Optional reading material will be posted every week.

Important information

- 10% late penalties per day, 0 mark after 5 days late.
- There is no exam.
- CUDA will be taught by Nvidia DLI, you will get a certificate if passed the test.
- Please post questions on discussion board.

Outline

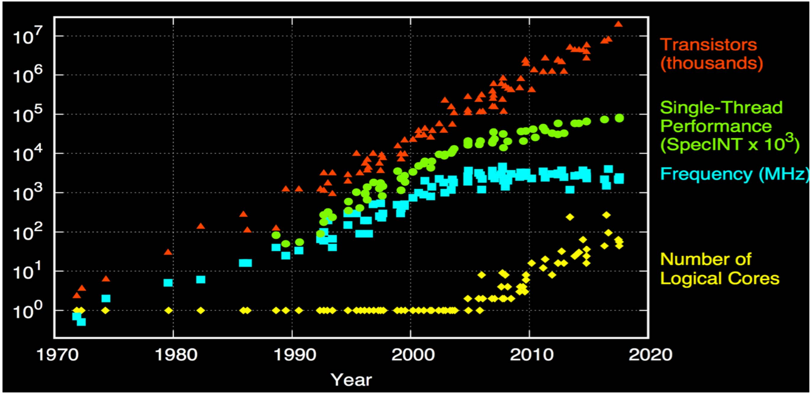
What is parallel computing?

Using multiple processors in parallel to solve problems more quickly than with a single processor.

Why parallel computing?

Why parallel computing?

- From 1986 – 2002, microprocessors were speeding like a rocket, increasing in performance an average of 50% per year.
- Since then, it's dropped to about 20% increase per year.



M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, C. Batten, and K. Rupp

Why parallel computing?

- Instead of designing and building faster microprocessors, put multiple processors on a single integrated circuit.
- Adding more processors doesn't help much if programmers aren't aware of them...
- ...or don't know how to use them.
- Serial programs don't benefit from this approach (in most cases).

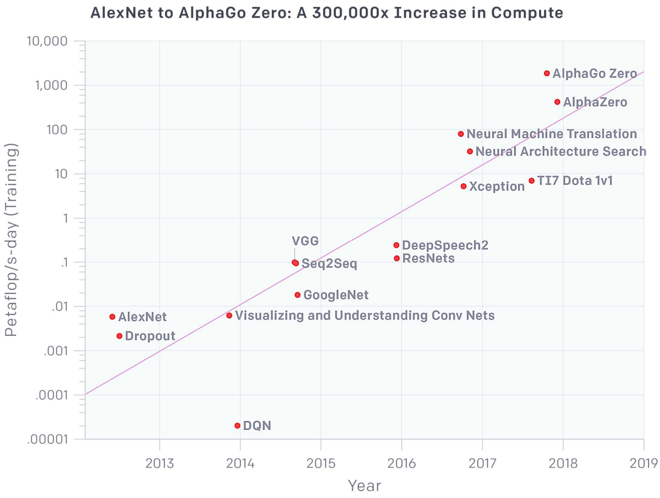
Why parallel computing?

- Computational power is increasing, but so are our computation problems and needs.
- Problems we never dreamed of have been solved because of past increases, such as decoding the human genome.
- More complex problems are still waiting to be solved.

Why parallel computing?

- Climate models, large-scale, more realistic simulations
- Machine Learning (deep learning)
- Bioinformatics
- Games! VR requires massive computational capabilities

Why parallel computing?



https://blog.openai.com/ai-and-compute/

Outline

Approaches to the serial problem

- Rewrite serial programs so that they're parallel.
- Write translation programs that automatically convert serial programs into parallel programs.
  - This is very difficult to do.
  - Success has been limited.

Example

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

```
sum = 0;

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

Example (cont.)

- We have p cores, p much smaller than n.
- 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 (cont.)

- After each core completes execution of the code, is a 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:
  1,4,3, 9,2,8, 5,1,1, 5,2,7, 2,5,0, 4,1,8, 6,5,1, 2,3,9

Example (cont.)

- 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.

Example (cont.)


```
if (I am 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 (cont.)

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

Global sum

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

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

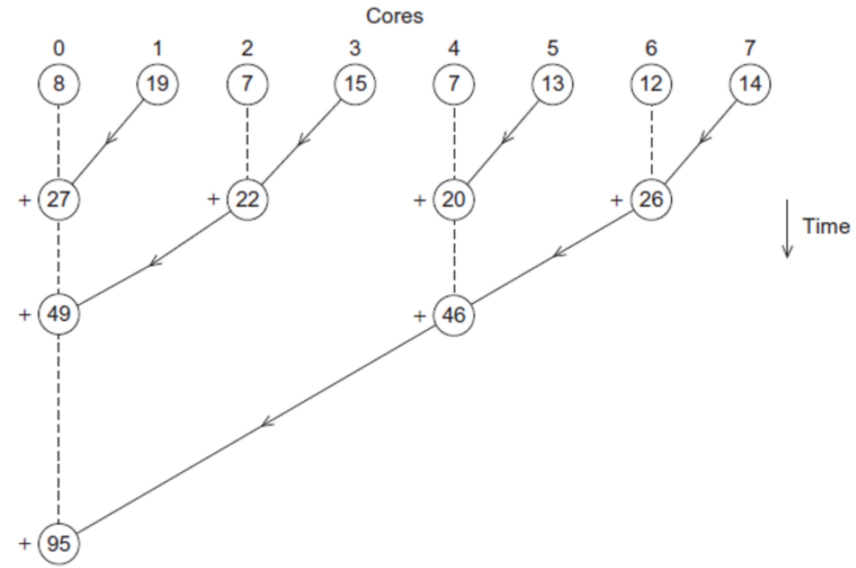
Better parallel algorithm

- Don't make the master core do all the work.
- Share it among the other cores.
- Pair the cores so that core 0 adds its result with core 1's result.
- Core 2 adds its result with core 3's result, etc.
- Work with odd and even numbered pairs of cores.

Better parallel algorithm (Cont.)

- Repeat the process now with only the evenly ranked cores.
- Core 0 adds result from core 2.
- Core 4 adds the 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



How do we write parallel programs?

- Task parallelism
  - Partition various 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.

Division of work – data parallelism

```
sum = 0;

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

Division of work – task parallelism


```
if (I am 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;
}
```

Coordination

- Cores usually need to coordinate their work.
- Communication – one or more cores send their current partial sums to another core.
- 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.

What we'll be doing

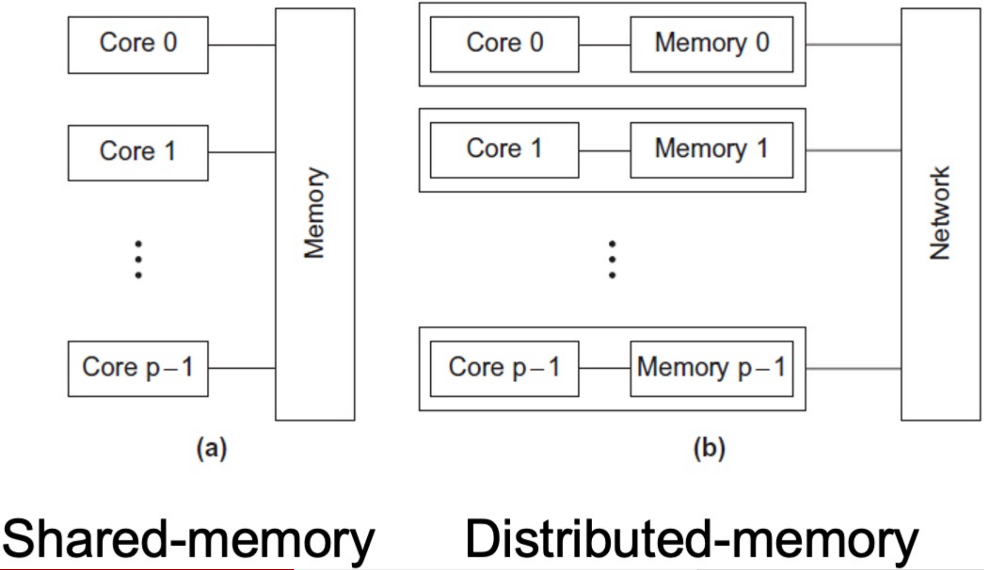
- Learning to write programs that are explicitly parallel.
- Using the C language.
- Using four different extensions to C.
  - Message-Passing Interface (MPI)
  - Posix Threads (Pthreads)
  - OpenMP
  - Nvidia CUDA

Outline

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 core has its own, private memory.
  - The cores must communicate explicitly by sending messages across a network.

Type of parallel systems



Terminology

- 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.
- Distributed computing – a program may need to cooperate with other programs to solve a problem.

Speedup and efficiency

- Speedup: Ratio of execution time on one process to that on p processes

$$Speedup = \frac{t_1}{t_p} \quad (1)$$

- Efficiency: Speedup per process

$$Efficiency = \frac{t_1}{t_p \times p} \quad (2)$$

- Speedup is limited by the serial portion of the code.
  - Often referred to as the serial "Bottleneck"
- Lets say only a fraction f of the code can be parallelised on p processes

$$Speedup = \frac{1}{(1-f)+f/p} \tag{3}$$

- Speedup is limited by the serial portion of the code.
  - Often referred to as the serial "Bottleneck"
- Lets say only a fraction f of the code can be parallelised on p processes

$$Speedup = \frac{1}{(1-f) + f/p} \tag{3}$$

$$Speedup = \frac{20s}{(20s - 18s) + 18s/p} \tag{4}$$

Summary

- Module introduction
- What is parallel computing?
- An parallelism example
- Terminology and definition

Next Week

- Designing Parallel Algorithms
- Distributed-memory programming using MPI