Tools Seminar

Week 10 - Parallel Computing

Hongzheng Chen
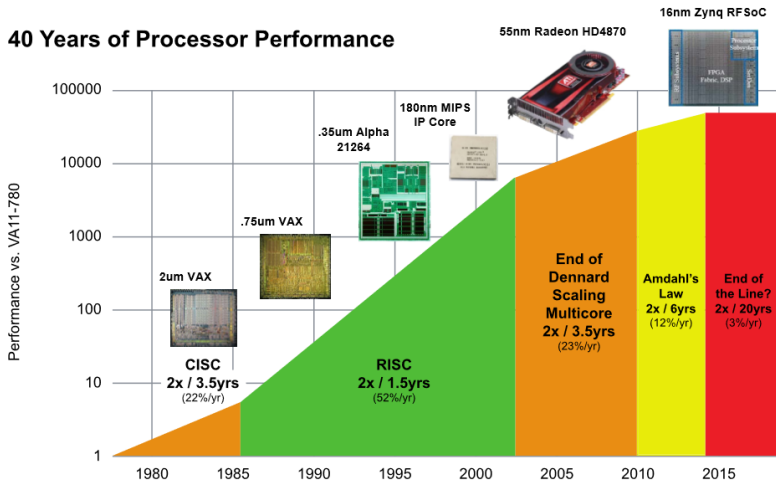
Mar 30, 2020

1

Introduction

Challenges: The End of Moore's Law and Scaling

40 Years of Processor Performance

Source: John Hennessy and David Patterson, Computer Architecture: A Quantitative Approach, 6/e 2018

* That's why Intel is called "toothpaste factory" now

The End of Moore's Law and Scaling

*This shift toward **increasing parallelism** is not a triumphant stride forward based on breakthroughs in novel software and architectures for parallelism; instead, this plunge into parallelism is actually a **retreat** from even greater challenges that thwart efficient silicon implementation of traditional uniprocessor architectures.*
— The Landscape of Parallel Computing Research: A View from Berkeley, 2006

The End of Moore's Law and Scaling

> *This shift toward **increasing parallelism** is not a triumphant stride forward based on breakthroughs in novel software and architectures for parallelism; instead, this plunge into parallelism is actually a **retreat** from even greater challenges that thwart efficient silicon implementation of traditional uniprocessor architectures.*
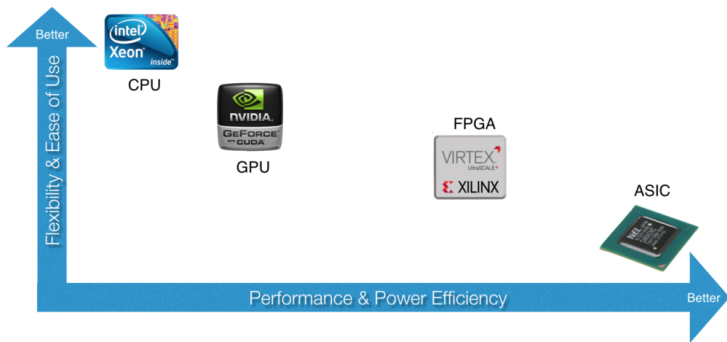> *— The Landscape of Parallel Computing Research: A View from Berkeley, 2006*
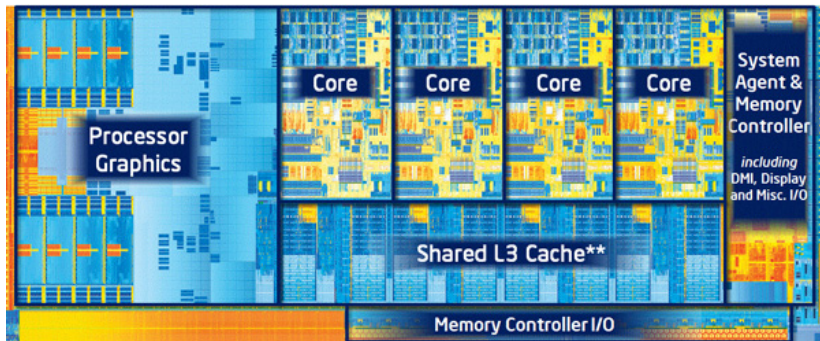
Thus, multicore processors put burdens from hardware to software, which needs programmers to code parallel programs.

Different hardware

- CPU
- GPU (Graphical Processing Unit)
- FPGA (Field-Programmable Gate Array)
- ASIC (Application-Specific Integrated Circuit)
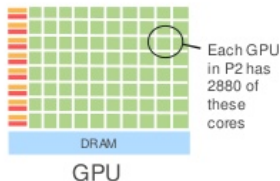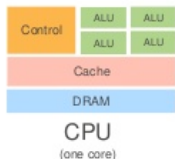
CPU Architecture

Intel core i7 CPU (Ivy Bridge)

Parallel Processing in GPUs and FPGAs

A GPU is effective at processing the <u>same set of operations</u> in parallel – single instruction, multiple data (SIMD). A GPU has a well-defined instruction-set, and fixed word sizes – for example single, double, or half-precision integer and floating point values.

Each GPU in P2 has 2880 of these cores

Each FPGA in F1 has more than 2M of these cells

CPU (one core)

GPU

FPGA

An FPGA is effective at processing the <u>same or different operations</u> in parallel – multiple instructions, multiple data (MIMD). An FPGA does not have a predefined instruction-set, or a fixed data width.

amazon webservices | Webinars
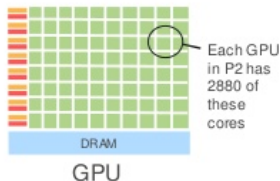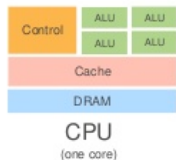
Parallel Processing in GPUs and FPGAs

A GPU is effective at processing the <u>same set of operations</u> in parallel – single instruction, multiple data (SIMD). A GPU has a well-defined instruction-set, and fixed word sizes – for example single, double, or half-precision integer and floating point values.

An FPGA is effective at processing the <u>same or different operations</u> in parallel – multiple instructions, multiple data (MIMD). An FPGA does not have a predefined instruction-set, or a fixed data width.

amazon webservices | Webinars
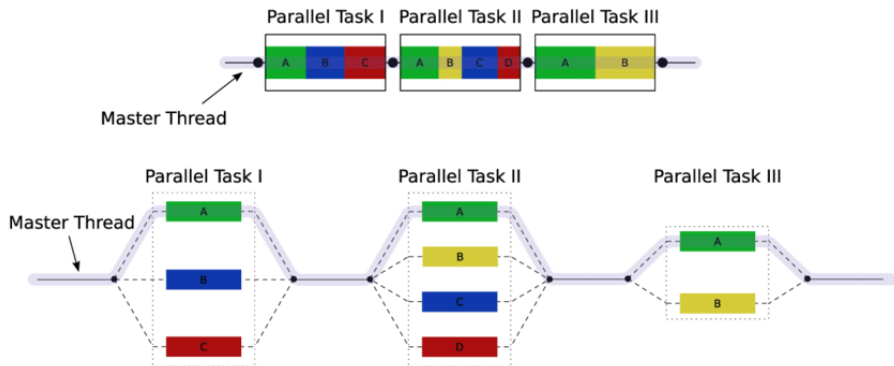
We will focus on CPU parallelism in this seminar

2

Shared-Memory Parallelism

2.1

Multi-threads

Fork-Join Model

- Fork: Dispatch tasks to each processor / thread
- Join: Synchronization, wait till all threads are done

pthread

POSIX (Portable Opearing System Interface for Unix)

- `<pthread.h>` is in Linux's system library and can be directly called

```
void *foo(void *arg)
{
    int* id = (int*) arg;
    printf ("My id is %d\n", *id);
}

int main()
{
    pthread_t  id [4];
    for (int i = 0; i < 4; ++i)
        // pass in function pointer and args
        pthread_create (&id[ i ], NULL,foo,&i);
    for (int i = 0; i < 4; ++i)
        pthread_join (&id[ i ], NULL);
    for (int i = 0; i < 4; ++i)
        pthread_exit (&id[ i ]) ;
}
```

Need to add `-lpthread` flag when compiling

C++11 thread

C++11 adds initial support for multi-threading in stl

```cpp
#include <iostream>
#include <thread>
using namespace std;

void exec(int n){
    cout << "My id is" << n << endl;
}

int main(){
    thread myThread[4];
    for (int i = 0; i < 4; ++i)
        myThread = thread(exec,i);
    for (int i = 0; i < 4; ++i)
        myThread[i].join();
}
```

Race Condition

Be careful of the shared data

Thread A	Thread B	Thread A		Thread B	
$\cdots$	$\cdots$	Load	Count	Load	Count
Count++	Count--	Add	#1	Sub	#1
$\cdots$	$\cdots$	Store	Count	Store	Count

- Critical section: That part of the program where the shared memory is accessed
- Need to avoid conflicts and make data consistent

Avoid Race Condition

Two basic methods:

- Corse-grained: Lock/mutex
- Fine-grained: Atomic operations

* There are lots of details about synchronization & consistency, please refer to books of OS

Mutex Operations in pthread

`<pthread.h>`

- `pthread_mutex_init(&mutex1,NULL)`
- `pthread_mutex_destroy(&mutex1)`
- `pthread_mutex_lock(&mutex1)`
- `pthread_mutex_unlock(&mutex1)`

`<thread>`

- `std::mutex g_display_mutex`
- `std::lock_guard<std::mutex> guard(g_display_mutex)`

Multi-threading in Python

- `threading.Thread`
- `multiprocessing.Process`
- `t1.start()`, `t1.join()`
- Global Interpreter Lock (GIL) limitation → CPython

    An interpreter that uses GIL always allows exactly one thread to execute at a time, even if run on a multi-core processor.

Ref: https://realpython.com/intro-to-python-threading/

2.2

OpenMP

OpenMP

OpenMP (Open Multi-Processing): Shared-memory programming model

- Set of parallel commands, library, and routines
- Simplify multi-threading programming
- A spec suitable for different devices from desktop to supercomputer
- gcc has initial support for OpenMP

OpenMP API

`#include <omp.h>` and only need to write compilation directives

```
#pragma omp <directive-name> [clause,...]
```

- `omp_get_thread_num`
- `omp_get/set_num_procs`
- `omp_get/set_num_threads`
- `#pragma omp parallel for`: The most commonly used!
- `#pragma omp ... private (<variable list>)`
- `#pragma omp ... reduction (op:list)`

OpenMP Example (Matrix Multiplication)

```c
# pragma omp for
for ( i = 0; i < n; i++ )
{
  for ( j = 0; j < n; j++ )
  {
    c[i][j] = 0.0;
    for ( k = 0; k < n; k++ )
    {
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
    }
  }
}
```

OpenMP Example (Summation)

```c
float sum(const float *a, size_t n)
{
    float total = 0.;

    #pragma omp parallel for reduction(+:total)
    for (size_t i = 0; i < n; i++) {
        total += a[i];
    }
    return total;
}
```

2.3

Cilk Plus

Intel Clik Plus

Intel Cilk Plus: A extremely light-weighted parallel framework

- `#include<cilk/cilk.h>`
- gcc 5.0+: `g++ -O3 -fcilkplus -lcilkrts <source>`
- Or compiled by Intel Compiler (icpc) — Better choice!
    - But from icpc 18.0, Intel uses Thread Building Block (TBB)

Only three keywords

- `cilk_spawn`: fork
- `cilk_sync`: join
- `cilk_for`: parallel for

Clik Example (Fibbonacci)

```
int fib(int n)
{
    if (n < 2)
        return n;
    int x = fib(n-1);
    int y = fib(n-2);
    return x + y;
}

int fib(int n)
{
    if (n < 2)
        return n;
    int x = cilk_spawn fib(n-1);
    int y = fib(n-2);
    cilk_sync;
    return x + y;
}
```

Cilk Runtime

The most powerful thing is Cilk runtime deploys **work-stealing** scheduling strategy, which greatly outperforms OpenMP's runtime
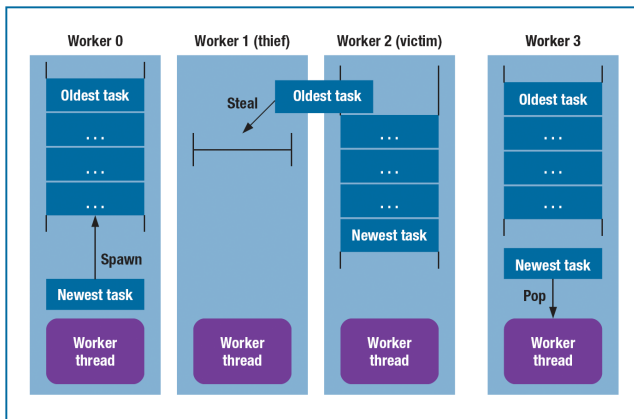

Fig source: Intel TBB

2.4

Finer-grained Parallelism

Parallelism

- Thread-Level Parallelism (TLP)
- Instruction-Level Parallelism (ILP)
  - Pipelining
  - Hyperscalar
  - Very Long Instruction Word (VLIW)
  - Vector processing
  - Out-of-Order (OoO) execution
  - Spectacular execution
- Data-Level Parallelism
  - SIMD (Single Instruction Multiple Data) array processor $\rightarrow$ GPU

* Please refer to Computer Architecture books / CSAPP

Intel CPU SIMD Instruction Set

- MME (Multi Media Extensions): Pentium, 1996
- SSE (Streaming SIMD Extensions): Pentium III, 1999
- AVX (Advanced Vector Extensions): Sandy Bridge 2008
- AVX2: Haswell, 2011

Naming Conventions

`_mm<bit_width>_<name>_<data_type>`

- `<bit_width>`: the return size, 128 - empty, 256 - 256
- `<name>`: describes the operation performed by the intrinsic
- `<data_type>`: the function's primary arguments

Instructions	Description
ps	packed single-precision
pd	packed double-precision
epi8/epi16/epi32/epi64	signed integers
epu8/epu16/epu32/epu64	unsigned integers
si128/si256	unspecified vector
m128/m128i/m128d	input vector types

e.g. `_mm256_srlv_epi64`: 64-bit signed int $\to$ 256-bit vector

AVX Example

```c
#include <immintrin.h>
#include <stdio.h>

int main() {

    /* Initialize the two argument vectors */
    __m256 evens = _mm256_set_ps(2.0, 4.0, 6.0, 8.0, 10.0, 12.0, 14.0, 16.0);
    __m256 odds = _mm256_set_ps(1.0, 3.0, 5.0, 7.0, 9.0, 11.0, 13.0, 15.0);

    /* Compute the difference between the two vectors */
    __m256 result = _mm256_sub_ps(evens, odds);

    /* Display the elements of the result vector */
    float* f = (float*) &result; // type conversion
     printf ("%f %f %f %f %f %f %f %f\n",
       f[0], f[1], f[2], f[3], f[4], f[5], f[6], f[7]);

    return 0;
}
```

Add -mavx flag when compiling

* Be careful that redundant movements reduce performance!

3

Distributed-Memory Parallelism

Message Passing Interface (MPI)

Need to install MPI compiler

- `#include<mpi.h>`
- `mpicc`, `mpic++`
- `mpirun -np 2 foo : -np 4 bar`

MPI Hello World

```c
#include <mpi.h>

int main(int argc, char* argv[])
{
    int npes, myrank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    printf("From process %d out of %d, Hello World!\n", myrank,
        ↪ npes);
    MPI_Finalize();
}
```

Message Passing

Message communication is the central part of MPI

```
MPI_Send(
    void* data,
    int count,
    MPI_Datatype datatype,
    int destination ,
    int tag,
    MPI_Comm communicator)

MPI_Recv(
    void* data,
    int count,
    MPI_Datatype datatype,
    int source ,
    int tag,
    MPI_Comm communicator,
    MPI_Status* status)
```

MPI Example Program (Ping Pong)

```c
int ping_pong_count = 0;
int partner_rank = (world_rank + 1) % 2;
while (ping_pong_count < PING_PONG_LIMIT) {
    if (world_rank == ping_pong_count % 2) {
        // Increment the ping pong count before you send it
        ping_pong_count++;
        MPI_Send(&ping_pong_count, 1, MPI_INT, partner_rank, 0,
                MPI_COMM_WORLD);
        printf("%d sent and incremented ping_pong_count "
                "%d to %d\n", world_rank, ping_pong_count,
                partner_rank);
    } else {
        MPI_Recv(&ping_pong_count, 1, MPI_INT, partner_rank, 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("%d received ping_pong_count %d from %d\n",
                world_rank, ping_pong_count, partner_rank);
    }
}
```

4

Parallel Computing Frameworks

Frameworks
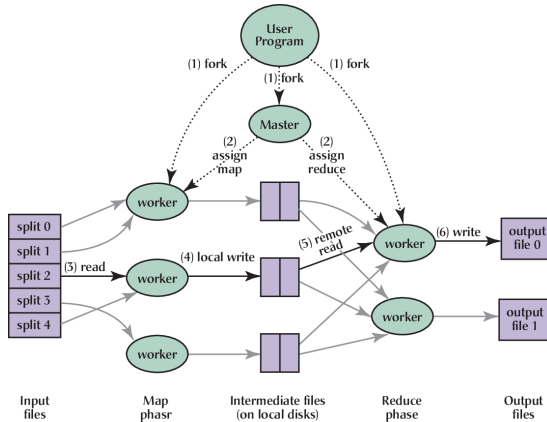
- MapReduce: Big data programming model → Hadoop
- Spark: Better data management
- Ray: Machine Learning

5

Summary

Summary

- Introduction to parallelism
- Shared-memory: pthreads, OpenMP, Cilk, AVX
- Distributed-memory: MPI
- Parallel computing frameworks: MapReduce