Parallel and Distributed Programming
Introduction

Kenjiro Taura

Contents
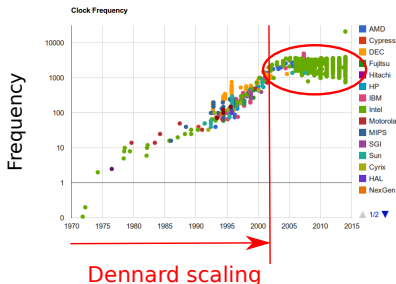
Contents

Why parallel?

- frequencies no longer increase (end of Dennard scaling)

source: http://cpudb.stanford.edu/

Why parallel?

- frequencies no longer increase (end of Dennard scaling)
- techniques to increase performance (Instruction-Level Parallelism, or ILP) of serial programs are increasingly difficult to pay off (Pollack's law)

source: http://cpudb.stanford.edu/
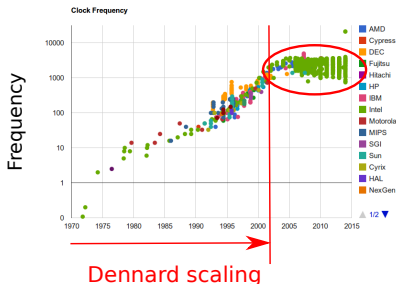
Why parallel?

- frequencies no longer increase (end of Dennard scaling)
- techniques to increase performance (Instruction-Level Parallelism, or ILP) of serial programs are increasingly difficult to pay off (Pollack's law)
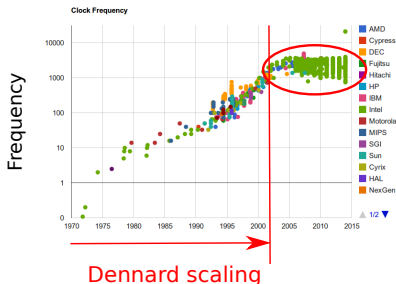- multicore, manycore, and GPUs are in part response to it

have more transistors? ⇒ have more cores

source: http://cpudb.stanford.edu/

There are no serial machines any more

- virtually all CPUs are now *multicore*
- high performance accelerators (GPUs and Xeon Phi) run at even low frequencies and have many more cores *(manycore)*

Processors for supercomputers are ordinary, perhaps even more so

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, **HPE** DOE/SC/Oak Ridge National Laboratory United States	8,730,112	1,102.00	1,685.65	21,100
2	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, **Fujitsu** RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
3	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, **HPE** EuroHPC/CSC Finland	1,110,144	151.90	214.35	2,942
4	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, **IBM** DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	200.79	10,096
5	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, **IBM / NVIDIA / Mellanox** DOE/NNSA/LLNL United States	1,572,480	94.64	125.71	7,438
6	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, **NRCPC** National Supercomputing Center in Wuxi China	10,649,600	93.01	125.44	15,371

https://www.top500.org/lists/top500/2022/06/

Implication to software

- existing serial SWs do not get (dramatically) faster on new CPUs

Implication to software

- existing serial SWs do not get (dramatically) faster on new CPUs
- just writing it in C/C++ goes nowhere close to machine's potential performance, unless you know how to exploit parallelism of the machine

Implication to software

- existing serial SWs do not get (dramatically) faster on new CPUs
- just writing it in C/C++ goes nowhere close to machine's potential performance, unless you know how to exploit parallelism of the machine
- you need to understand
  - does it use multiple cores (and how the work is distributed)?
  - does it use SIMD instructions?
  - does it have good instruction level parallelism?
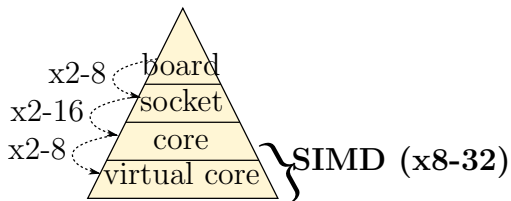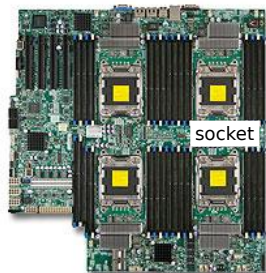
Example: matrix multiply

- how much can we improve this on a single machine?

```
void gemm(long M, long N, long K,
          float A[M][K], float B[K][N], float C[M][N]) {
  long i, j, k;
  for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
      for (k = 0; k < K; k++)
        C[i][j] += A[i][k] * B[k][j];
}
```

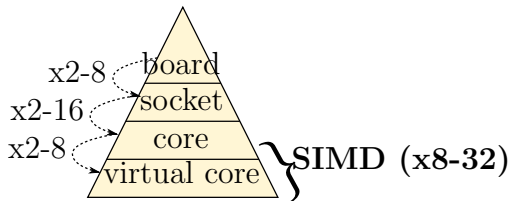# Contents

What a single parallel machine (node) looks like

- SIMD : Single Instruction Multiple Data
- a single SIMD register holds many values
- a single instruction applies the same operation (e.g., add, multiply, etc.) on all data in a SIMD register
- a single core can execute multiple instructions in each cycle (ILP)

What a single parallel machine (node) looks like

- SIMD : Single Instruction Multiple Data
- a single SIMD register holds many values
- a single instruction applies the same operation (e.g., add, multiply, etc.) on all data in a SIMD register
- a single core can execute multiple instructions in each cycle (ILP)

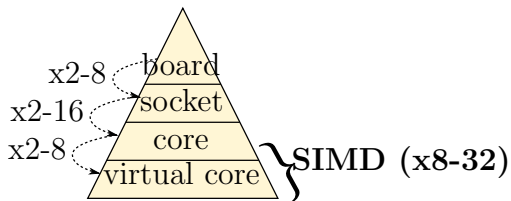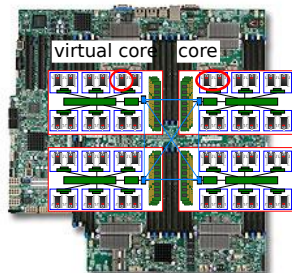# What a single parallel machine (node) looks like

- SIMD : Single Instruction Multiple Data
- a single SIMD register holds many values
- a single instruction applies the same operation (e.g., add, multiply, etc.) on all data in a SIMD register
- a single core can execute multiple instructions in each cycle (ILP)

What a machine looks like
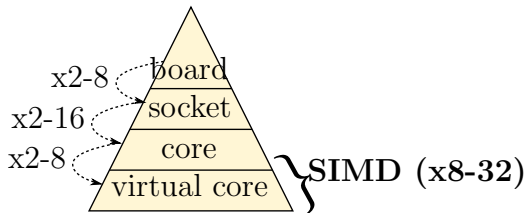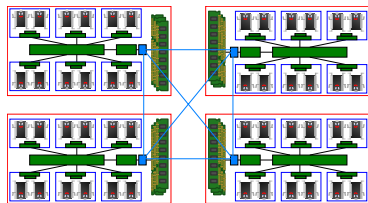
- performance comes from *multiplying* parallelism of many levels
- parallelism (per CPU)

    = SIMD width × instructions/cycle × cores

- in particular, peak FLOPS (per CPU)

    = (2 × SIMD width) × FMA insts/cycle/core × freq × cores

- FMA: Fused Multiply Add ($d = a * b + c$)
- the first factor of 2 : multiply and add (each counted as a flop)

What a GPU looks like?

Streaming Multiprocessor

- a GPU consists of many *Streaming Multiprocessors (SM)*
- each SM is highly multithreaded and can interleave many *warps*
- each warp consists of 32 *CUDA threads*; in a single cycle, threads in a warp can execute the same single instruction

What a GPU looks like?

- despite very different terminologies, there are more commonalities than differnces

GPU	CPU
SM	core
multithreading in an SM	simultaneous multithreading
a warp (32 CUDA threads)	a thread executing SIMD instructions
	multiple instructions from a single thread

- there are significant differeces too, which we'll cover later

How much parallelism?

- Intel CPUs

arch model	SIMD width SP/DP	FMAs /cycle /core	freq GHz	core	peak GFLOPS SP/DP	TDP W
Haswell E78880Lv3	8/4	2	2.0	18	1152/576	115
Broadwell 2699v4	8/4	2	2.2	22	1548/604	145
Cascade Lake 9282	16/8	2	2.6	56	9318/4659	400
Ice Lake 8368	16/8	2	2.4	38	5836/2918	270

- NVIDIA GPUs (numbers are without Tensor Cores)

acrh model	threads /warp	FMAs /cycle /SM SP/DP	freq GHz	SM	paek GFLOPS SP/DP	TDP W
Pascal P100	32	2/1	1.328	56	9519/4760	300
Volta V100	32	2/1	1.530	80	15667/7833	300
Ampere A100	32	2/1	1.410	108	19353/9676	400

Peak (SP) FLOPS

Ice Lake 8368

$$= (2 \times 16) \text{ [flops/FMA insn]}$$
$$\times \ 2 \text{ [FMA insns/cycle/core]}$$
$$\times \ 2.4\text{G [cycles/sec]}$$
$$\times \ 38 \text{ [cores]}$$
$$= 5836 \text{ GFLOPS}$$

A100

$$= (2 \times 32) \text{ [flops/FMA insn]}$$
$$\times \ 2 \text{ [FMA insns/cycle/SM]}$$
$$\times \ 1.41\text{G [cycles/sec]}$$
$$\times \ 108 \text{ [SMs]}$$
$$= 19353 \text{ GFLOPS}$$

NVIDIA: Tensor Cores

- performance shown so far is limited by the fact that a single (FMA) instruction can perform 2 flops (1 multiply + 1 add)
- Tensor Core, a special execution unit for a small matrix-multiply-add, changes that
- A100's each Tensor Core can do $C = A \times B + C$ (where $A : 4 \times 4$, $B : 4 \times 8$) per cycle ($A : 4 \times 4$ TF32, $B : 4 \times 8$ TF32, C and D are SP)

$$2 \times 4 \times 4 \times 8 = 256 \text{ flops/cycle}$$

- each SM of A100 GPU has 4 Tensor Cores, so a single A100 device can do

$$
\begin{aligned}
& (2 \times 4 \times 4 \times 8) \text{ [flops/cycle]} \\
\times \quad & 1.41\text{G [cycles/sec]} \\
\times \quad & 4 \times 108 \text{ [Tensor Cores]} \\
= \quad & 155934.72 \text{ GFLOPS}
\end{aligned}
$$

Trends

- processors' performance improvement is getting less and less "generic" or "transparent"
  - frequencey + instruction level parallelism
    - → explicit parallelism (multicore/manycore)
    - → special execution unit for macro operations (e.g., MMA)
    - → application-specific instructions (?)
- performance is getting more and more dependent on programming

Contents

So how to program it?

- no matter how you program it, you want to maximally utilize all forms of parallelism
- "how" depends on devices and programming languages

Language constructs for multiple cores / GPUs

from low level to high levels

- (CPU) OS-level threads
- (GPU) CUDA threads
- SPMD $\approx$ the entire program runs with N threads
- parallel loops
- dynamically created tasks
- internally parallelized libraries (e.g., matrix operations)
- high-level languages executing pre-determined operations (e.g., matrix operations, map & reduce-like patterns, deep learning) in parallel

Language constructs for CPU SIMD

from low level to high levels

- assembly
- intrinsics
- vector types
- vectorized loops
- internally vectorized libraries (e.g., matrix operations)

This lecture is for . . .

those who want to:

- have a first-hand experience in parallel and high performance programming (OpenMP, CUDA, SIMD, . . .)

This lecture is for . . .

those who want to:

- have a first-hand experience in parallel and high performance programming (OpenMP, CUDA, SIMD, . . .)
- know good tools to solve more complex problems in parallel (divide-and-conquer and task parallelism)

This lecture is for . . .

those who want to:

- have a first-hand experience in parallel and high performance programming (OpenMP, CUDA, SIMD, . . .)
- know good tools to solve more complex problems in parallel (divide-and-conquer and task parallelism)
- understand when you can get "close-to-peak" CPU/GPU performance and how to get it (SIMD and instruction level parallelism)

This lecture is for . . .

those who want to:

- have a first-hand experience in parallel and high performance programming (OpenMP, CUDA, SIMD, . . .)
- know good tools to solve more complex problems in parallel (divide-and-conquer and task parallelism)
- understand when you can get "close-to-peak" CPU/GPU performance and how to get it (SIMD and instruction level parallelism)
- learn many reasons why you don't get good parallel performance

This lecture is for . . .

those who want to:

- have a first-hand experience in parallel and high performance programming (OpenMP, CUDA, SIMD, . . .)
- know good tools to solve more complex problems in parallel (divide-and-conquer and task parallelism)
- understand when you can get "close-to-peak" CPU/GPU performance and how to get it (SIMD and instruction level parallelism)
- learn many reasons why you don't get good parallel performance
- have a good understanding about caches and memory and why they matter so much for performance