Why parallel?

- frequencies no longer increase (end of Dennard scaling)
Why parallel?

- frequencies no longer increase (end of Dennard scaling)
- techniques to increase performance (ILP) of serial programs are increasingly difficult to pay off (Pollack’s law)
Why parallel?

- frequencies no longer increase (end of Dennard scaling)
- techniques to increase performance (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
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)*
Supercomputers look ordinary, perhaps more so

- Sunway ($\approx 1.45$GHz)
- Xeon Phi ($\approx 1$GHz)
- NVIDIA GPU ($< 1$GHz)
- CPUs running at $\approx 2.0$GHz

---

<table>
<thead>
<tr>
<th>Rank</th>
<th>Site</th>
<th>System</th>
<th>Cores</th>
<th>Rmax (TFlop/s)</th>
<th>Rpeak (TFlop/s)</th>
<th>Power (kW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>National Supercomputing Center in Wuxi, China</td>
<td>Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C</td>
<td>10,649,600</td>
<td>93,014.6</td>
<td>125,435.9</td>
<td>15,371</td>
</tr>
<tr>
<td>2</td>
<td>National Super Computer Center in Guangzhou, China</td>
<td>Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.2GHz, TH</td>
<td>3,120,000</td>
<td>33,862.7</td>
<td>54,902.4</td>
<td>17,808</td>
</tr>
<tr>
<td>3</td>
<td>DOE/SC/Oak Ridge National Laboratory, United States</td>
<td>Titan - Cray XK7, Opteron 6274 16C 2.2GHz, Cray Gemini Interconnect, NVIDIA K20x Cray Inc.</td>
<td>560,640</td>
<td>17,590.0</td>
<td>27,112.5</td>
<td>8,209</td>
</tr>
<tr>
<td>4</td>
<td>DOE/NNSA/LLNL, United States</td>
<td>Sequoia - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM</td>
<td>1,572,864</td>
<td>17,173.2</td>
<td>20,132.7</td>
<td>7,890</td>
</tr>
<tr>
<td>5</td>
<td>RIKEN Advanced Institute for Computational Science (AICS), Japan</td>
<td>K computer, SPARC64 VIIIfx 2.0GHz, Tofu Interconnect Fujitsu</td>
<td>705,024</td>
<td>10,510.0</td>
<td>11,280.4</td>
<td>12,660</td>
</tr>
<tr>
<td>6</td>
<td>DOE/SC/Argonne National Laboratory, United States</td>
<td>Mira - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM</td>
<td>786,432</td>
<td>8,586.6</td>
<td>10,066.3</td>
<td>3,945</td>
</tr>
<tr>
<td>7</td>
<td>DOE/NNSA/LANL/SNL, United States</td>
<td>Trinity - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Aries</td>
<td>301,056</td>
<td>8,100.9</td>
<td>11,078.9</td>
<td>3,458</td>
</tr>
</tbody>
</table>

[www.top500.org](http://www.top500.org)
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?
  - if so, how work is distributed?
  - does it use SIMD instructions (covered later)?
Example: matrix multiply

Q: how much can we improve this on my laptop?

```c
void gemm(long n, /* n = 2400 */
        float A[n][n], float B[n][n], float C[n][n]) {
    long i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                C[i][j] += A[i][k] * B[k][j];
}
```
Q: how much can we improve this on my laptop?

```c
void gemm(long n, /* n = 2400 */
  float A[n][n], float B[n][n], float C[n][n]) {
  long i, j, k;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      for (k = 0; k < n; k++)
        C[i][j] += A[i][k] * B[k][j];
}
```

$ ./simple_mm
C[1200][1200] = 3011.114014
in 56.382360 sec
2.451831 GFLOPS
Example: matrix multiply

Q: how much can we improve this on my laptop?

```c
void gemm(long n, /* n = 2400 */
    float A[n][n], float B[n][n], float C[n][n]) {
    long i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                C[i][j] += A[i][k] * B[k][j];
}
```

```
$ ./simple_mm
C[1200][1200] = 3011.114014
in 56.382360 sec
2.451831 GFLOPS
```

```
$ ./opt_mm
C[1200][1200] = 3011.108154
in 1.302980 sec
106.095263 GFLOPS
```
Contents

1 Why Parallel Programming?

2 What Parallel Machines Look Like, and Where Performance Come From?

3 How to Program Parallel Machines?

4 How to Program Parallel Machines?
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
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
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
performance comes from multiplying parallelism of many levels
parallelism (per CPU)
\[= \text{SIMD width} \times \text{instructions/cycle} \times \text{cores}\]
in particular, peak FLOPS (per CPU)
\[= (2 \times \text{SIMD width}) \times \text{FMA insts/cycle/core} \times \text{freq} \times \text{cores}\]
FMA: Fused Multiply Add \((d = a \times b + c)\)
the first factor of 2: multiply and add (each counted as a flop)
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 differences

<table>
<thead>
<tr>
<th>GPU</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>SM</td>
<td>core</td>
</tr>
<tr>
<td>multithreading in an SM</td>
<td>simultaneous multithreading</td>
</tr>
<tr>
<td>a warp</td>
<td>a thread executing SIMD instructions</td>
</tr>
<tr>
<td>CUDA thread</td>
<td>each lane of a SIMD instruction</td>
</tr>
</tbody>
</table>

- there are significant differences too, which we’ll cover later
## How much parallelism?

### Intel CPUs

<table>
<thead>
<tr>
<th>arch model</th>
<th>SIMD width SP/DP</th>
<th>FMAs /cycle /core</th>
<th>freq GHz</th>
<th>core</th>
<th>peak GFLOPS SP/DP</th>
<th>TDP W</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haswell E78880Lv3</td>
<td>8/4</td>
<td>2</td>
<td>2.0</td>
<td>18</td>
<td>1152/576</td>
<td>115</td>
</tr>
<tr>
<td>Broadwell 2699v4</td>
<td>8/4</td>
<td>2</td>
<td>2.2</td>
<td>22</td>
<td>1548/604</td>
<td>145</td>
</tr>
<tr>
<td>Skylake 6130</td>
<td>16/8</td>
<td>2</td>
<td>2.1</td>
<td>16</td>
<td>2150/1075</td>
<td>125</td>
</tr>
<tr>
<td>KNM 7285</td>
<td>16/8</td>
<td>2</td>
<td>1.4</td>
<td>68</td>
<td>6092/3046</td>
<td>250</td>
</tr>
</tbody>
</table>

### NVIDIA GPUs

<table>
<thead>
<tr>
<th>archh model</th>
<th>threads /warp</th>
<th>FMAs /cycle /SM SP/DP</th>
<th>freq GHz</th>
<th>SM</th>
<th>peak GFLOPS SP/DP</th>
<th>TDP W</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pascal P100</td>
<td>32</td>
<td>2/1</td>
<td>1.328</td>
<td>56</td>
<td>9519/4760</td>
<td>300</td>
</tr>
<tr>
<td>Volta v100</td>
<td>32</td>
<td>2/1</td>
<td>1.530</td>
<td>80</td>
<td>15667/7833</td>
<td>300</td>
</tr>
</tbody>
</table>
Peak (SP) FLOPS

Skylake 6130

\[
= (2 \times 16) \text{[flops/FMA insn]} \times 2 \text{[FMA insns/cycle/core]} \times 2.1G \text{[cycles/sec]} \times 28 \text{[cores]}
\]

= 2150 GFLOPS

Volta V100

\[
= (2 \times 32) \text{[flops/FMA insn]} \times 2 \text{[FMA insns/cycle/SM]} \times 1.53G \text{[cycles/sec]} \times 80 \text{[SMs]}
\]

= 15667 GFLOPS
Contents

1. Why Parallel Programming?

2. What Parallel Machines Look Like, and Where Performance Come From?

3. How to Program Parallel Machines?

4. How to Program Parallel Machines?
So how to program it?

- no matter how you program it, you want to maximally utilize multiple cores and SIMD instructions
- “how” depends on programming languages
Language constructs for multiple cores

from low level to high levels

- OS-level 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 and map & reduce-like patterns) in parallel (Torch7, Chainer, Spark, etc.)
Language constructs for 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, TBB, SIMD, . . .)
This lecture is for . . .

those who want to:

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