CS420 – Lecture 1

Raghavendra Kanakagiri
Slides: Marc Snir

Spring 2023

Logistics

CS420/ECE492/CSE402 – Parallel Programming for Science & Engineering – 3 and 4 units

	Name	Office	Office hours	email
Instructor	Raghavendra Kanakagiri (Raghu)	SC 4215		raghaven@illinois.edu
TA	Santhosh Mohan			sm107@illinois.edu

Who the course is for:

- People that need to develop parallel codes – especially for scientific computations
- Focused on practical skills needed to achieve better performance via parallelism

- Piazza: `https://piazza.com/class/lcw7m4hxv8w2es`
    - Lecture slides will be posted on piazza.
- Quizzes, homeworks, grades will be on Gradescope
- MPs on Gitlab.

Grading

Type	%
MPs	35%
HWs	20%
Midterm	20%
Final Exam	25%

- If taking 4 point option then above determines 75% of grade and final project determines 25%
- *All (but the final project) require individual work. You can discuss an MP before starting to program, but you program on your own.*
  - See The CS Dept Honor Code at `http://cs.illinois.edu/academics/honor-code`
- All late submissions will be graded for 70% points.

Introduction

Outline

parallel algorithms Only briefly covered in this course. See CS 498

parallel architecture Briefly covered, so as to understand performance bottlenecks of parallel systems. See also CS 533

parallel programming Main focus of course; see also CS 483/ ECE 408, CS 484

Outline

- Acquire basic knowledge of CPU architecture: execution pipeline, dependencies, caches; learn to tune performance by enhancing locality and leveraging compiler optimizations.
- Understand vector instructions and learn to use vectorization
- Acquire basic knowledge of multicore architectures: cache coherence, true and false sharing and their relevance to parallel performance tuning
- Learn to program using multithreading, parallel loops, and multitasking using a language such as OpenMP. Learn to avoid concurrency bugs.
- Learn to program using message passing with a library such as MPI.
- Understand simple parallel algorithms and their complexity.
- Learn to program accelerators using a language such as OpenMP.
- Acquire basic understanding of parallel I/O and of frameworks for data analytics, such as map-reduce.

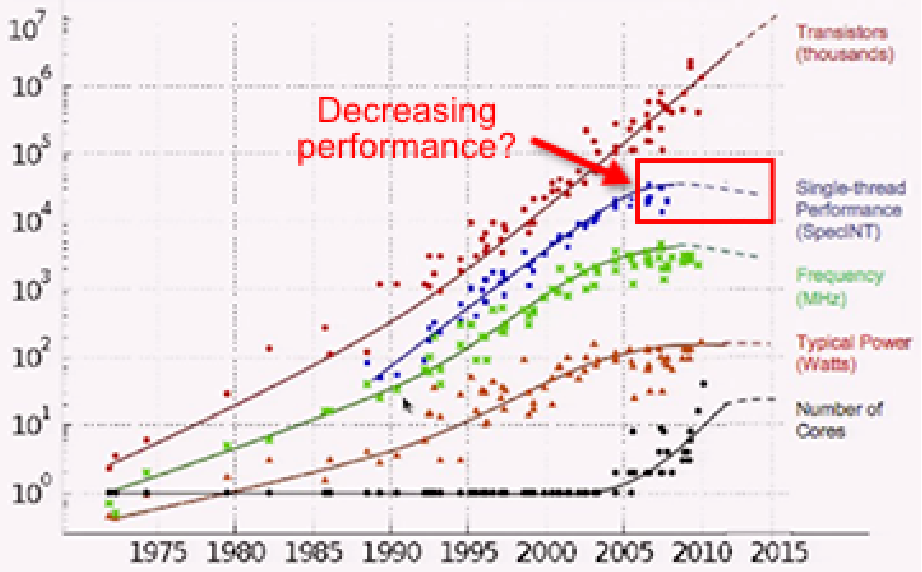
Outline

- Computer organization: Pipelining, vector instructions and memory hierarchy
- Shared memory programming: OpenMP
- Distributed memory programming: MPI
- Accelerators: GPUs: OpenMP

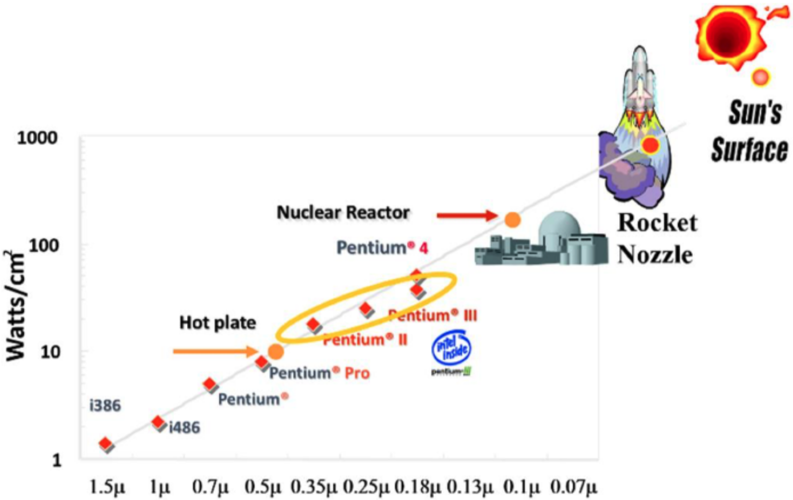
Outline

- Prerequisite: CS 225
- **Not** CS 233 (Computer architecture) or CS 241 (Systems programming)
- *Prepare <u>non-CS</u> science and engineering students to the use of parallel computing in support of their work. See CS 484.*

Waiting for computers to become faster is not an option



Waiting for computers to become faster is not an option



Technology Evolution

- Number of transistors per chip continues to increase (likely to stop in a few years)

Technology Evolution

- Number of transistors per chip continues to increase (likely to stop in a few years)
- $\Rightarrow$ Cannot increase anymore power consumption of chip – cooling

Technology Evolution

- Number of transistors per chip continues to increase (likely to stop in a few years)
- $\Rightarrow$ Cannot increase anymore power consumption of chip – cooling
- $\Rightarrow$ Therefore, cannot increase clock speed

Technology Evolution

- Number of transistors per chip continues to increase (likely to stop in a few years)
- $\Rightarrow$ Cannot increase anymore power consumption of chip – cooling
- $\Rightarrow$ Therefore, cannot increase clock speed
- Instructions per Cycle (IPC) - instruction level parallelism provides diminishing returns.

Technology Evolution

- Number of transistors per chip continues to increase (likely to stop in a few years)
- $\Rightarrow$ Cannot increase anymore power consumption of chip – cooling
- $\Rightarrow$ Therefore, cannot increase clock speed
- Instructions per Cycle (IPC) - instruction level parallelism provides diminishing returns.
- $\Rightarrow$ Only road to faster execution is explicit program parallelism

Levels of parallelism

Single Instruction, Multiple Data (SIMD): One instruction processes a vector of operands. E.g., (Intel) 512 bits = 16 single precision (32 bit) words, or 8 double precision words.

Levels of parallelism

Single Instruction, Multiple Data (SIMD): One instruction processes a vector of operands. E.g., (Intel) 512 bits $=$ 16 single precision (32 bit) words, or 8 double precision words.

Shared memory parallelism: Multiple physical threads, each executing its own instruction stream, run simultaneously. E.g. Intel Xeon Phi up to 72 cores; each core runs up to 4 simultaneous threads; $72\times4=288$.

Levels of parallelism

Single Instruction, Multiple Data (SIMD): One instruction processes a vector of operands. E.g., (Intel) 512 bits = 16 single precision (32 bit) words, or 8 double precision words.

Shared memory parallelism: Multiple physical threads, each executing its own instruction stream, run simultaneously. E.g. Intel Xeon Phi up to 72 cores; each core runs up to 4 simultaneous threads; $72 \times 4 = 288$.

- Threads within core share compute resources; threads in distinct cores only share memory

Levels of parallelism

Single Instruction, Multiple Data (SIMD): One instruction processes a vector of operands. E.g., (Intel) 512 bits = 16 single precision (32 bit) words, or 8 double precision words.

Shared memory parallelism: Multiple physical threads, each executing its own instruction stream, run simultaneously. E.g. Intel Xeon Phi up to 72 cores; each core runs up to 4 simultaneous threads; $72 \times 4 = 288$.

- Threads within core share compute resources; threads in distinct cores only share memory
- 288 is number of *simultaneous* physical threads. System may have thousands of *concurrent* software threads, but they do not run all the time.

Levels of parallelism

Single Instruction, Multiple Data (SIMD): One instruction processes a vector of operands. E.g., (Intel) 512 bits $=$ 16 single precision (32 bit) words, or 8 double precision words.

Shared memory parallelism: Multiple physical threads, each executing its own instruction stream, run simultaneously. E.g. Intel Xeon Phi up to 72 cores; each core runs up to 4 simultaneous threads; $72\times4=288$.

- Threads within core share compute resources; threads in distinct cores only share memory
- 288 is number of *simultaneous* physical threads. System may have thousands of *concurrent* software threads, but they do not run all the time.

Distributed Memory Parallelism: Many processors (nodes) are connected together with a fast network; parallel application can utilize many nodes at once. E.g., Fugaku has 7,630,848 cores and peak performance of 537 Pflop/s.

Petaflop/s $= 10^{15}$ floating point operations per second.

Frontier

ORNL's Frontier First to Break the Exaflop Ceiling

May 30, 2022

The 59th edition of the TOP500 revealed the Frontier system to be the first true exascale machine with an HPL score of 1.102 Exaflop/s.

The No. 1 spot is now held by the Frontier system at Oak Ridge National Laboratory (ORNL) in the US. Based on the latest HPE Cray EX235a architecture and equipped with AMD EPYC 64C 2GHz processors, the system has 8,730,112 total cores, a power efficiency rating of 52.23 gigaflops/watt, and relies on gigabit ethernet for data transfer.

read more »

- `top500.org`

- `top500.org`
- Frontier:
  - 9,472 AMD Epyc 7A53s "Trento" 64 core 2 GHz CPUs (606,208 cores) and 37,888 Radeon Instinct MI250X GPUs (8,335,360 cores)
  - 1.102 exaFLOPS

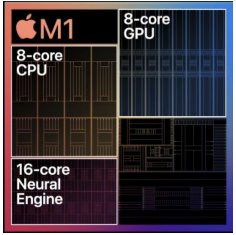
- `top500.org`
- Frontier:
  - 9,472 AMD Epyc 7A53s "Trento" 64 core 2 GHz CPUs (606,208 cores) and 37,888 Radeon Instinct MI250X GPUs (8,335,360 cores)
  - 1.102 exaFLOPS
- Fugaku:
  - 158,976 nodes Fujitsu A64FX CPU (48 cores)

- `top500.org`
- Frontier:
  - 9,472 AMD Epyc 7A53s "Trento" 64 core 2 GHz CPUs (606,208 cores) and 37,888 Radeon Instinct MI250X GPUs (8,335,360 cores)
  - 1.102 exaFLOPS
- Fugaku:
  - 158,976 nodes Fujitsu A64FX CPU (48 cores)
- Sunway TaihuLight
  - 64kB of scratchpad memory for data

- `top500.org`
- Frontier:
  - 9,472 AMD Epyc 7A53s "Trento" 64 core 2 GHz CPUs (606,208 cores) and 37,888 Radeon Instinct MI250X GPUs (8,335,360 cores)
  - 1.102 exaFLOPS
- Fugaku:
  - 158,976 nodes Fujitsu A64FX CPU (48 cores)
- Sunway TaihuLight
  - 64kB of scratchpad memory for data
- `https://www.exascaleproject.org`

- top500.org
- Frontier:
  - 9,472 AMD Epyc 7A53s "Trento" 64 core 2 GHz CPUs (606,208 cores) and 37,888 Radeon Instinct MI250X GPUs (8,335,360 cores)
  - 1.102 exaFLOPS
- Fugaku:
  - 158,976 nodes Fujitsu A64FX CPU (48 cores)
- Sunway TaihuLight
  - 64kB of scratchpad memory for data
- https://www.exascaleproject.org
- Apple's M1?

- top500.org
- Frontier:
  - 9,472 AMD Epyc 7A53s "Trento" 64 core 2 GHz CPUs (606,208 cores) and 37,888 Radeon Instinct MI250X GPUs (8,335,360 cores)
  - 1.102 exaFLOPS
- Fugaku:
  - 158,976 nodes Fujitsu A64FX CPU (48 cores)
- Sunway TaihuLight
  - 64kB of scratchpad memory for data
- https://www.exascaleproject.org
- Apple's M1?



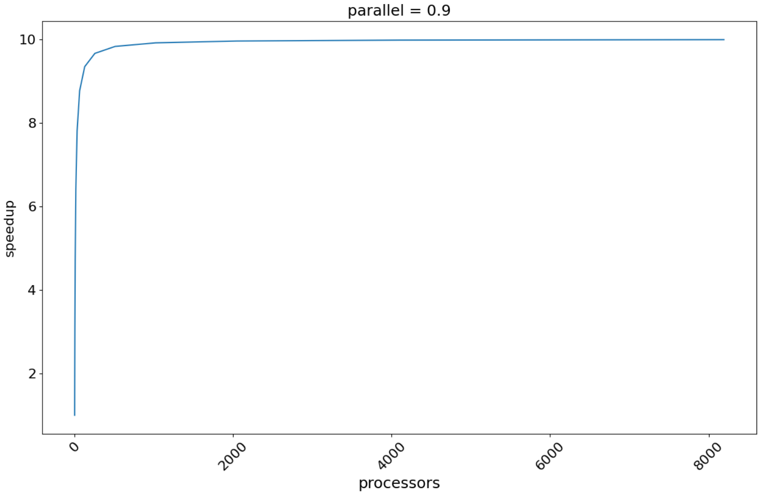
Parallel computing

- Speedup
- What is the overall speedup if you make 90% of the program 1000 times faster?

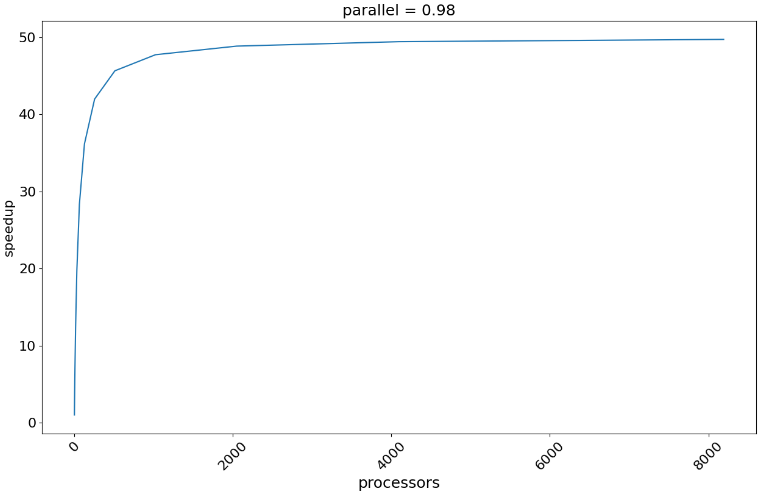
Parallel computing

- Speedup
- What is the overall speedup if you make 90% of the program 1000 times faster?
- $Speedup_{overall} = \frac{1}{(1 - frac_{enchanced}) + \frac{frac_{enchanced}}{Speedup}}$

Amdahl's Law


parallel = 0.9
speedup
10
8
6
4
2
0
2000
4000
6000
8000
processors

Amdahl's Law



parallel = 0.98
speedup
processors
0
10
20
30
40
50
0
2000
4000
6000
8000

Single Thread Performance

What is “performance”

Compute time required to solve a problem:

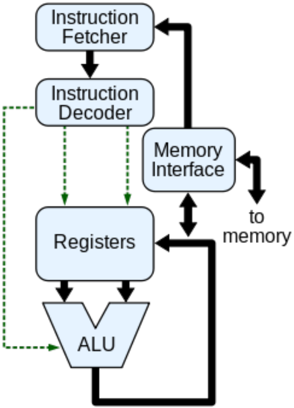
- Wall-clock time (assuming dedicated system)
- CPU time (time shared system)

- Depends on problem
- Depends on input size
- Depends on algorithm and code used
- Depends on system used (compiler, libraries, hardware)

Floating point rate

- For much of scientific computing one cares about floating point operations (flop)
- Can define *floating point rate* as number of flops per second achieved (for a particular code, input size, system, etc.).
- HPL: maximum floating point rate achieved for LU decomposition (direct solver for dense matrix) on a problem as large as the user wants: LINPACK benchmark used for Top500.
  - Best achieved (Fugaku), Rmax = 442 PFlop/s
- HPCG: maximum flop rate achieved by another benchmark code (focused on Conjugate Gradient, sparse matrices)
  - Best achieved (Fugaku), Rmax=16 Pflop/s

A simple processor



Those five status could be parallelize. Example of building house.

Assembly

Five status of a single processor: Fetch the instructions -> Decode the instructions ->

source code

```
temp = a[0];
for (i=1; i<N; i++) {
  temp = s+temp;
  a[i]=temp
}
```

g++ -g -O0 -Wa,-aslh code.cc

```
main:
mov     r2, #28
str     lr, [sp, #-4]!
sub     sp, sp, #108
add     r3, sp, #4
add     r1, sp, #100
.L2:
str     r2, [r3, #4]!
cmp     r3, r1
add     r2, r2, #27
bne     .L2
ldr     r0, [sp, #100]
bl      exit
```