CS420 – Lecture 1

Marc Snir

Fall 2018

ADMINISTRATION

WELCOME

Who we are:

	Name	Office	Office housr	email
Instructor	Marc Snir	SC 4232	Wed 2-3 pm	snir@illinois.edu
TA	Omri Mor	SC 0207	Mon 2-3 pm	omrimor2@illinois.edu
TA	Shibi He	SC 0207	Thu ??	shibihe@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
- **Not** for CS majors
- Graduates are required to take 4 units (do final project); undergraduates may also, if they wish so. (Need to change registration.)

Curriculum

The course will discuss approaches to improving program performance by

- Leveraging compilers and tools
- Improving locality
- Using vector operations and pipelining
- Using shared memory parallelism with OpenMP
- Using GPU accelerators with OpenMP
- Using distributed memory parallelism with MPI
- Using map-reduce frameworks

Course requires C++, C or Fortran

- Course is full and there are people who could not register: If you decide to drop the course, please do so ASAP
- Slides are posted on piazza: piazza.com/illinois/fall2018/cs420cse402ece492/home
  - Lecture slides are posted before the lecture and (usually) corrected after the lecture
- Lecture recordings are posted on Echo: echo360.org
- Quizzes, homeworks, grades will be on Compass
- MPs are submitted using GIT

Grading

Type	Frequency	%
Quizzes	every 2 weeks	5%
MPs	every 3 weeks	25%
Midterm Exam	(Tentative: 10/12)	25%
Final Exam		35%

- 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`
- No credit for late assignments

INTRODUCTION

Related Research Areas

Related Research Areas

Distributed Computing Multiple systems cooperating to solve a *loosely coupled* problem; systems can be geographically distributed. Examples: WWW, SETI@home

Related Research Areas

Distributed Computing Multiple systems cooperating to solve a *loosely coupled* problem; systems can be geographically distributed. Examples: WWW, SETI@home

Concurrent Computing Coordination of independent activities that use shared resources; system is often *reactive*. Could run on a single processor. Usually *nondeterministic*. Examples: Online Transaction Processing, OS

Related Research Areas

Distributed Computing Multiple systems cooperating to solve a *loosely coupled* problem; systems can be geographically distributed. Examples: WWW, SETI@home

Concurrent Computing Coordination of independent activities that use shared resources; system is often *reactive*. Could run on a single processor. Usually *nondeterministic*. Examples: Online Transaction Processing, OS

Parallel Computing The use of multiple threads in order to speed up a computation; system is usually *transformative*. Can be *deterministic* – nondeterminism can be part of the solution, but is not usually, part of the problem

Related Research Areas

Distributed Computing Multiple systems cooperating to solve a *loosely coupled* problem; systems can be geographically distributed. Examples: WWW, SETI@home

Concurrent Computing Coordination of independent activities that use shared resources; system is often *reactive*. Could run on a single processor. Usually *nondeterministic*. Examples: Online Transaction Processing, OS

Parallel Computing The use of multiple threads in order to speed up a computation; system is usually *transformative*. Can be *deterministic* – nondeterminism can be part of the solution, but is not usually, part of the problem

Division is fuzzy: *Cloud Computing* combines aspects of distributed computing and parallel computing

Topics Include

Topics Include

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

Topics Include

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

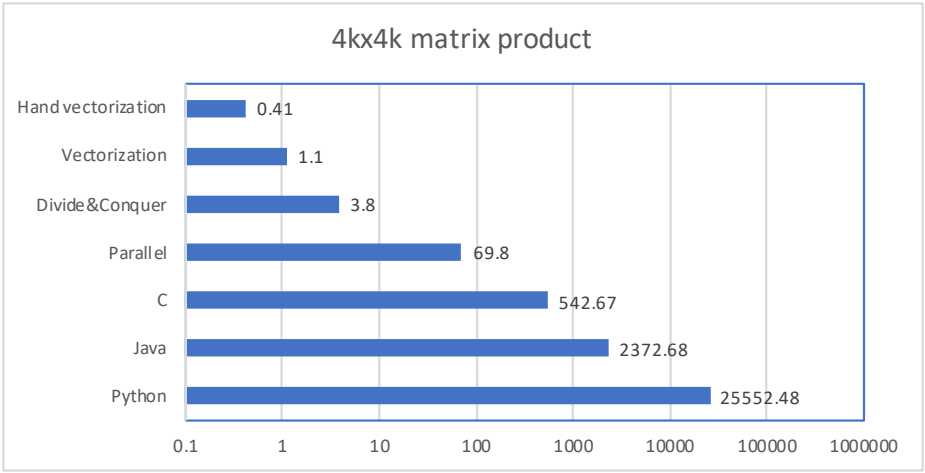
Topics Include

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

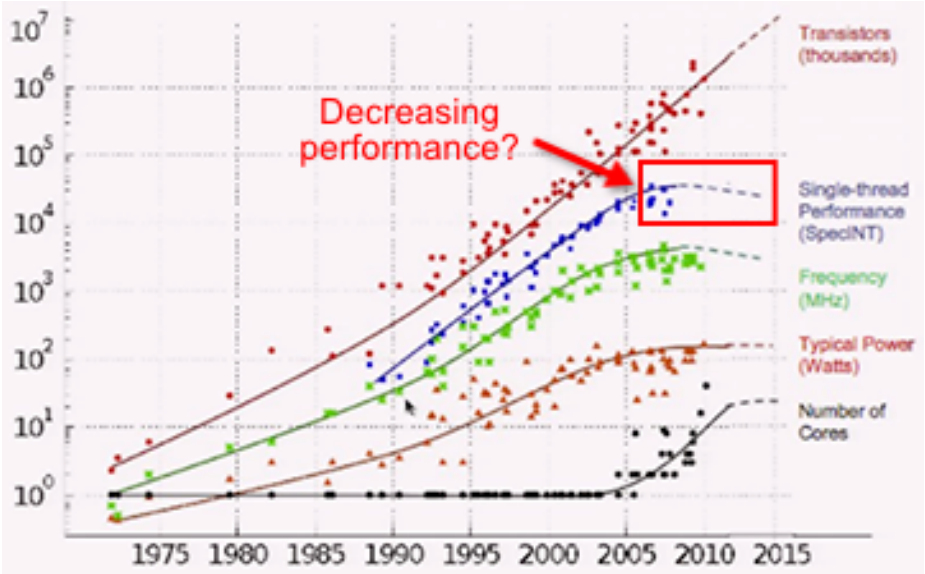
parallel programming Main focus of course; see also CS 483/ ECE 408, CS 484

Do you care about performance?



Waiting for computers to become faster is not an option

Waiting for computers to become faster is not an option



Technology Evolution

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 (unchanged since 2004)

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 (unchanged since 2004)
- Cannot increase 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 (unchanged since 2004)
- Cannot increase Instructions per Cycle (IPC) - instruction level parallelism provides diminishing returns.
- $\Rightarrow$ Only road to faster execution is explicit program parallelism

Levels of 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; $72 \times 4 = 288$.

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
- 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; $72 \times 4 = 288$.

- 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.

Distributed Memory Parallelism: Many processors (nodes) are connected together with a fast network; parallel application can utilize many nodes at once. E.g., Summit (Current top supercomputer) has 2,282,544 cores and peak performance of 187 Pflop/s.

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; 72×4=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.

Distributed Memory Parallelism: Many processors (nodes) are connected together with a fast network; parallel application can utilize many nodes at once. E.g., Summit (Current top supercomputer) has 2,282,544 cores and peak performance of 187 Pflop/s.

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

- kilo 10^3
- mega 10^6
- giga 10^9
- tera 10^{12}
- peta 10^{15}
- exa 10^{18}

- mili 10^{-3}
- micro 10^{-6}
- nano 10^{-9}
- pico 10^{-12}
- fento 10^{-15}

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)

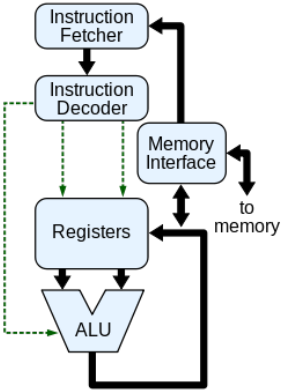
Floating point rate

- 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 (Summit), Rmax = 122.3 Pflop/s
- HPCG: maximum flop rate achieved by another benchmark code (focused on Conjugate Gradient, sparse matrices)
  - Best achieved (Summit), Rmax=2.9 Pflop/s (2% of HPL on same machine)

Uninteresting metrics

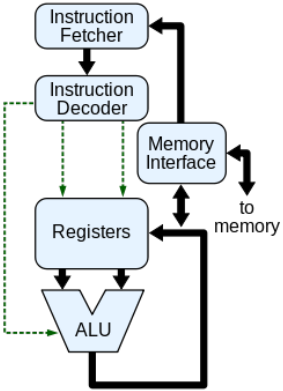
- *Peak performance:* the maximum flop rate the machine can achieve theoretically.
- AKA *Never to be exceeded performance*
- *Efficiency:* The ratio between achieved flop rate and peak performance.
- Efficiency is a terrible misnomer: A less "efficient" system can actually provide better performance/cost ratio (be more efficient in reality)
- It is important to use efficiently the expensive components of the system, not the cheap ones.

A simple processor



- Instructions executed sequentially
- Execution is *pipelined*: ALU executes instruction i while decoder decodes instruction $i + 1$ and fetcher fetches instruction $i + 2$ (simplified view).

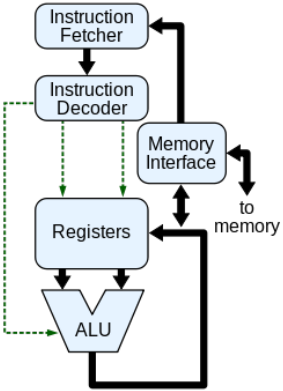
A simple processor



- Instructions executed sequentially
- Execution is *pipelined*: ALU executes instruction i while decoder decodes instruction $i + 1$ and fetcher fetches instruction $i + 2$ (simplified view).

  Problem: Branches break the pipeline

A simple processor

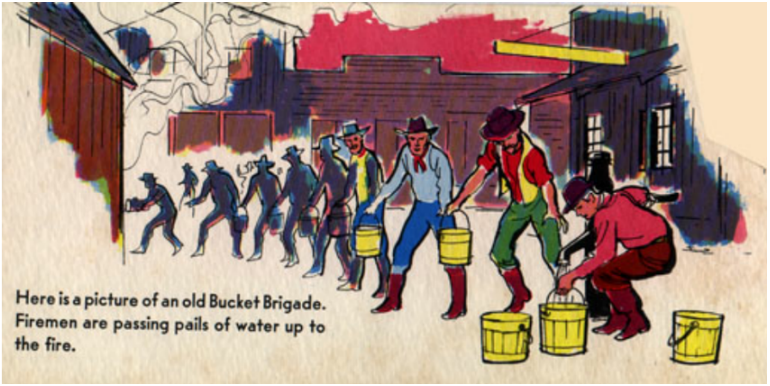


- Instructions executed sequentially
- Execution is *pipelined*: ALU executes instruction i while decoder decodes instruction $i + 1$ and fetcher fetches instruction $i + 2$ (simplified view).

  Problem: Branches break the pipeline

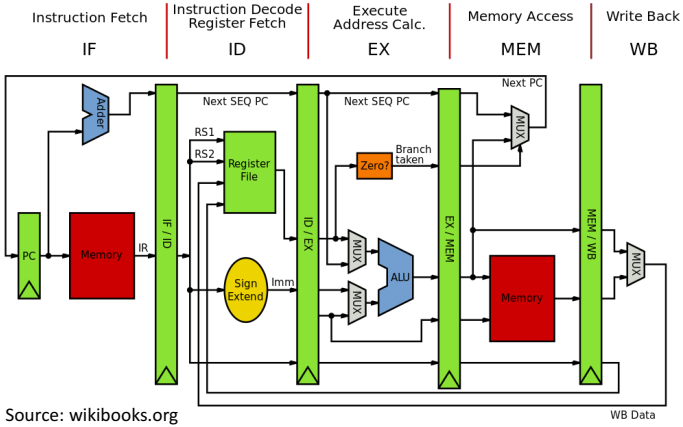
  Partial solution: Branch prediction

Pipelining



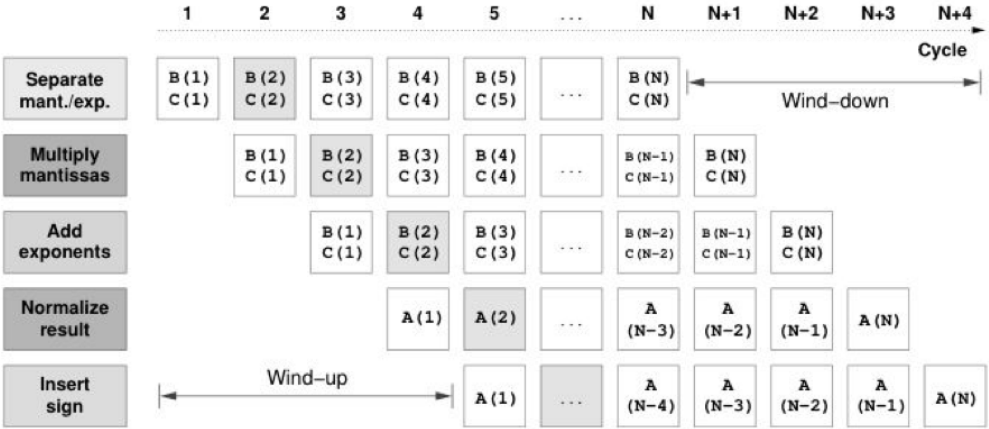
- Pipeline increases *throughput* – buckets per second
- Pipeline does not decrease *latency* (slightly increases it) – seconds for bucket transport from source to destination
- Pipeline enables more specialization (assembly line)

Reducing Gate Delays
Pipelined Processor



Source: wikibooks.org

Floating-point pipeline



Memory pipeline

- Can have multiple load/stores being handled simultaneously

Dependencies cause pipeline bubbles

Dependencies cause pipeline bubbles

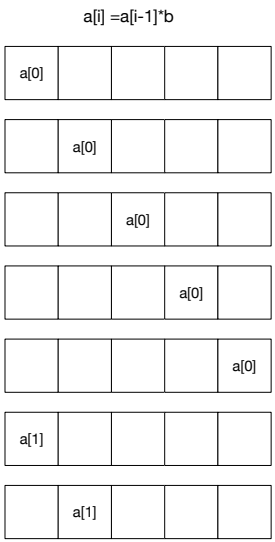
- *pipeline bubble:* empty stage in pipeline

Dependencies cause pipeline bubbles

- *pipeline bubble:* empty stage in pipeline
- Compute `a[i]=a[i-1]*s;` assume only constrained resource is floating point pipeline.

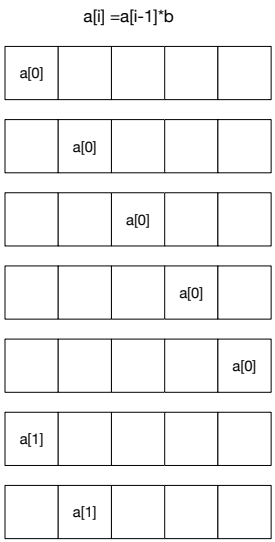
Dependencies cause pipeline bubbles

- *pipeline bubble:* empty stage in pipeline
- Compute `a[i]=a[i-1]*s`; assume only constrained resource is floating point pipeline.
- Dependency distance one implies only one stage of the pipeline is utilized (utilization $1/m$, if pipeline has depth m)

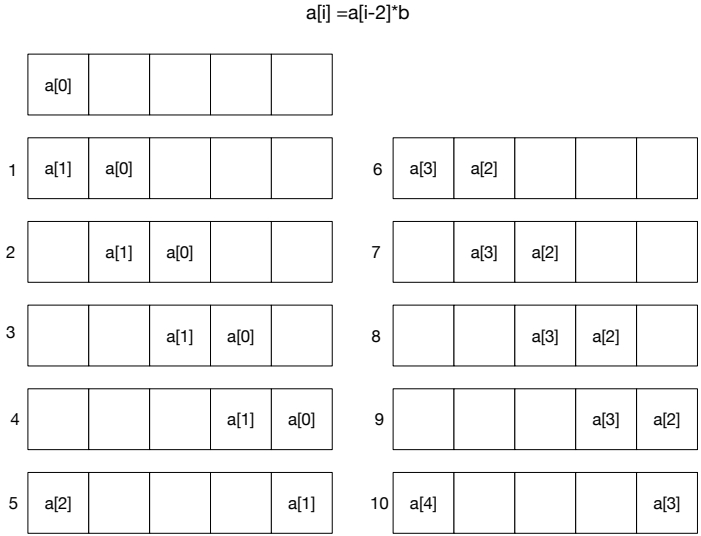


Dependencies cause pipeline bubbles

- *pipeline bubble:* empty stage in pipeline
- Compute `a[i]=a[i-1]*s`; assume only constrained resource is floating point pipeline.
- Dependency distance one implies only one stage of the pipeline is utilized (utilization $1/m$, if pipeline has depth m)

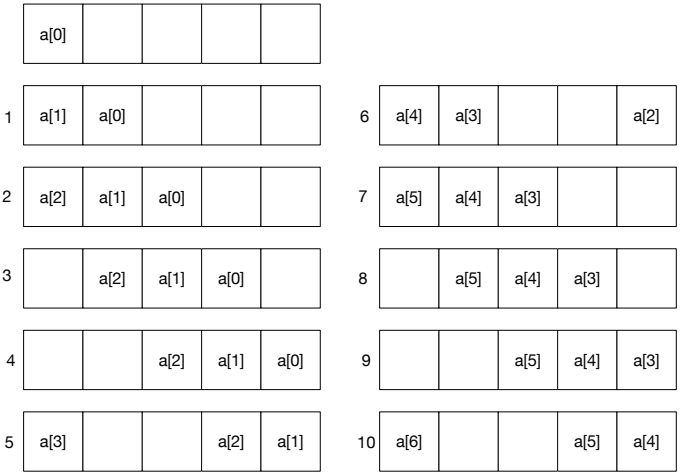


Dependency distance 2, utilization $2/m$



Dependency distance 3, utilization $3/m$; ...

a[i] =a[i-3]*b



True Dependences

```
for(i=2;i<N;i++)
a[i]=s*a[i-2];
```

- *Loop dependence*: iteration i depends on iteration $i-2$ (dependence distance 2).
- $\Rightarrow$ Can compute at same time iteration i and $i-1$, but no more.
- Need to optimize code, in order to compute simultaneously two iterations

```
for(i=2;i<N;i+=2) {
a[i]=s*a[i-2];
a[i+1]=s*a[i-1]
}
if (N%2) a[N-1]=s*a[N-3]
```

- More efficient code: Fewer branches, can pipeline (but need extra code to handle odd N); good if N is large
- Compiler will do this *loop unrolling* on its own, if possible (with higher optimization levels)
- *True Dependence:* Read after write (RAW) - producer/consumer dependence – Later iteration uses result of previous iteration.

False dependences 1

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

- *Anti dependence:* Write after read (WAR) – Later iteration updates variable used by earlier iteration. Seems to prevent pipelining
- Not "true dependence" – can be avoided by using temporary variables:

```
for(i=0;i<N;i+=2) {
t1=a[i+1];
t2 =a[i+2];
a[i] = s*t1;
a[i+1]=s*t2;
}
```

- Compiler will do this, using registers as temporary variables
- Code can be unrolled 3,4,... times; but, if unroll too much, may not have enough registers (register pressure)

False dependences 2

```
for (i=0;i<N;i++)
s=a[i];
```

- *Output dependence:* Write after write (WAW) –Updates need to occur in the right order
- Can be (usually) avoided by not performing first write:

```
s=a[N-1];
```

Control dependence

```
for(i=0;i<N;i++)
if(a[i][0] ==key) {
value = a[i][1];
break
}
```

- Whether iteration $i+1$ executes depends on the result of the test in iteration i.
- Can start, speculatively, to execute iteration $i+1$ – e.g., load `a[i+1][*]` – as long as speculation can be undone, if wrong.
- Processors do *branch prediction* and speculate on the most likely branch