A quick revision

Aravind Sukumaran Rajam

What happened?

- Travel Issues
- Thank fully Prof. Ananth stepped in

What are we going to do today?

- Why is this course important?
- How will this help us in our future (academic / industry)?
- My vision for this course
- What I promise
- What I want you to promise me
- A quick overview of the syllabus
- A quick revision of what you have learned
- Data dependencies
- Introduction to OpenMP

Why is this course important?

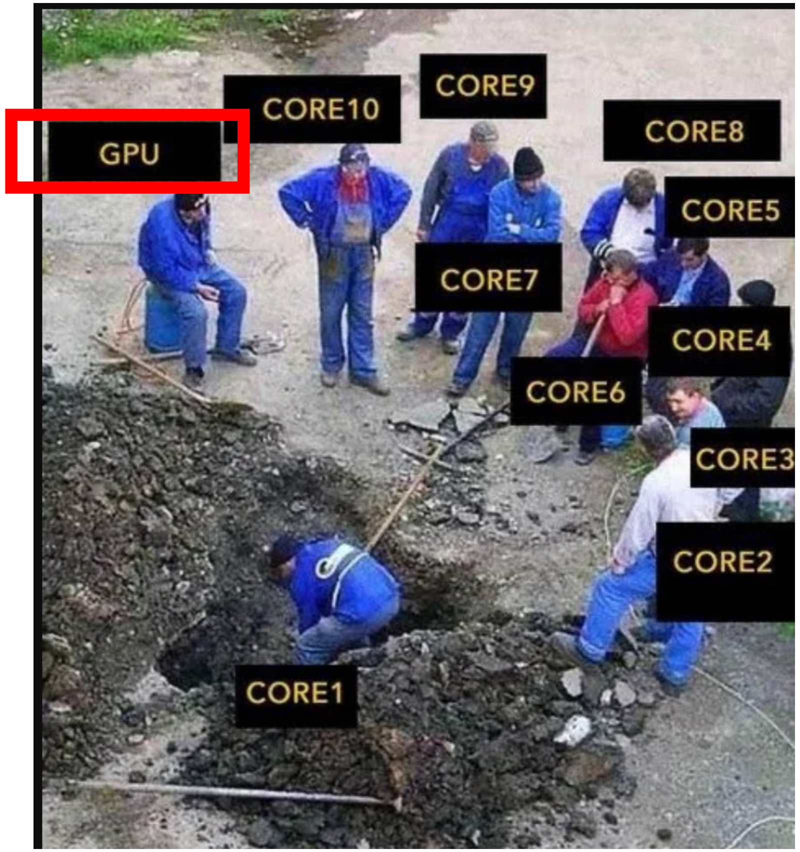
- There is a saying that “Mathematics is the language of science”
- Today, we can tell that “High Performance Computing (HPC)” is the beating heart of computer science, along with machine learning and computer security
  - Parallel computing is at the core of HPC
- Nothing is sequential anymore… Why? (we will see soon)
- Parallel computing knowledge is a basic requirement now a days
- Machine learning, scientific computing, video games, computer simulations, all relies on parallel computing

Why is this course important?

- Tremendous opportunities in the industry
- Knowing parallel computing will help you a lot in academic and industry research
  - BTW you should do a PhD or Masters (more on this in the future)
  - Want to try HPC? This course is page 1 ☺
- You will definitely become a better programmer
- Understand the hardware better
- Improve algorithmic knowledge

Gems!!!

- This is what will happen if you don't have parallel computing knowledge



Credits: To the unknown creator

My vision for this course

- My overall vision for this course is to make sure that you **understand** the basics of Parallel Computing
    - There is a big difference between scoring in an exam and understanding something
- To get you interested in HPC
- To get you interested in HPC related Graduate Studies

What I promise

- I will take every effort to make sure that you **understand** what I teach
- I will actively encourage you to think openly
- I will make every effort to accommodate your feedbacks into the course
  - Not next year!!! This year itself!!!
  - If I can't accommodate a request, I will tell you the reason
  - You will be provided plenty of opportunities to provide anonymous feedbacks

What I want you to promise me

- After every lecture, the same day you will at least spent 15 minutes to think about what we have learned

What I want you to promise me

Doing this is not going to help you!!!

Credits: To the unknown creator

What I want you to promise me

- After every lecture, the same day you will at least spent 15 minutes to think about what we have learned
- You will not copy assignments
- You will be active in the class and ask a lot of questions
  - There are no stupid questions
- You will actively give feedbacks to me
- You will put every effort to succeed in this course

A quick overview of the syllabus

```
int val = 3 / 2 * 4;
printf("%d", val);
```

a) 6
b) 0.375
c) 4
d) NA

Do you know how many people have made this mistake

A quick overview of the syllabus

- Introduction to fundamental parallel computing concepts
- A quick review of computer architecture (parallel units and caches)
- Data dependence analysis
- Introduction to shared memory parallelization:
    - Introduction to OpenMP
    - Task level parallelization
    - Loop level parallelization
- Advanced OpenMP
    - Parallel algorithms
    - Challenges associated with sparse computations
- Performance monitoring and debugging
- Introduction to distributed computing
- Introduction to MPI programming
- Advanced MPI
    - Visualization and performance monitoring
    - Advanced MPI API's
    - Challenges associated with MPI
- To infinity and beyond
    - What are the limitations of OpenMP and MPI
    - Can we design better parallel protocols
    - Future courses

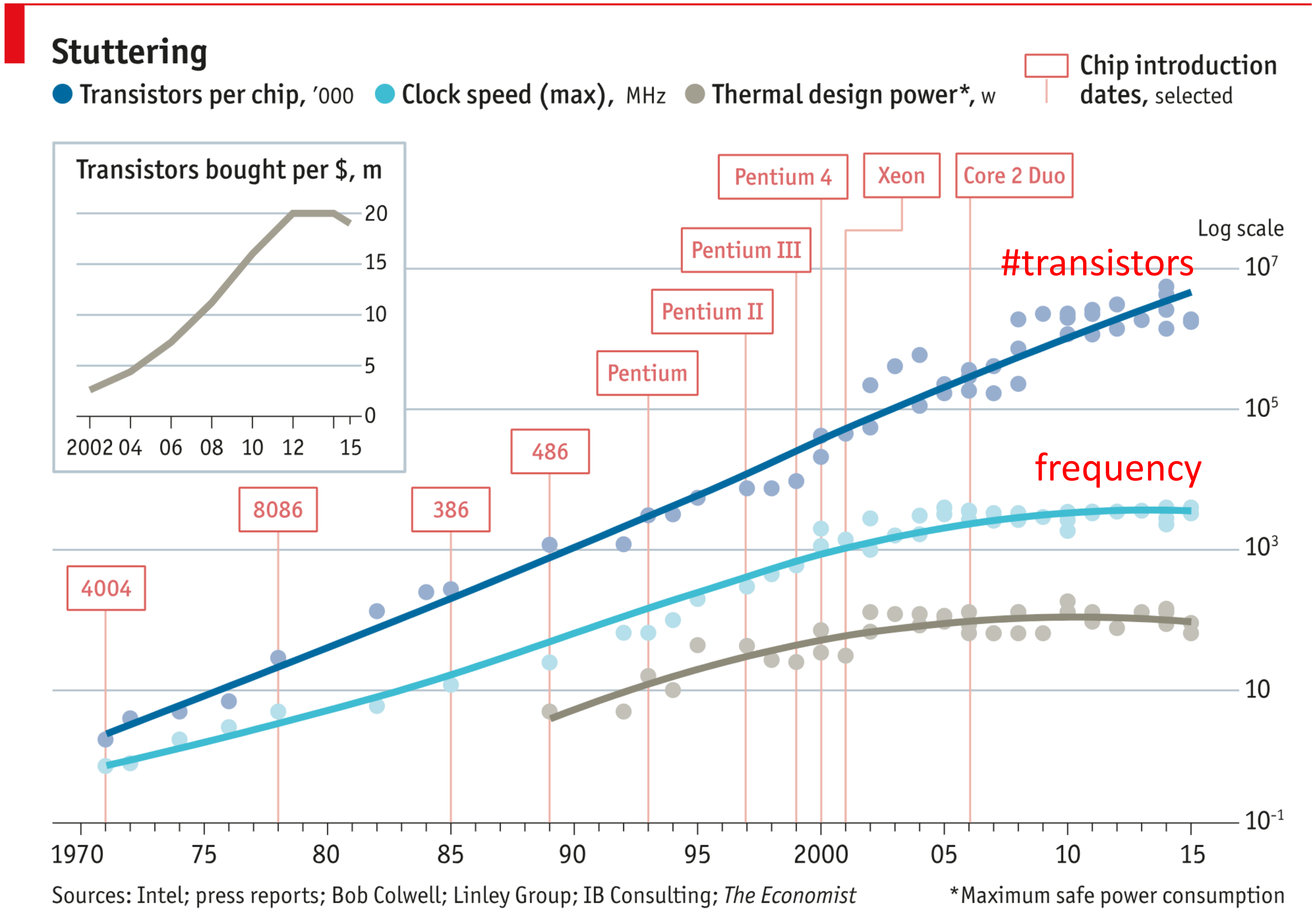
A quick revision of what you have learned

- Why parallel computing?
  - Why not go with sequential computing

End of Moore's Law

Problem:

- How to achieve good **performance**?
- Increase clock frequency?
  - stalled around 2000*
- Transistor count?
  - Nearing physical limits
  - 5nm? (maybe 3nm)



A quick revision of what you have learned

- Why parallel computing?
  - Why not go with sequential computing
  - What are the advantages?
  - What are the disadvantages?

Parallel computing

Advantages?

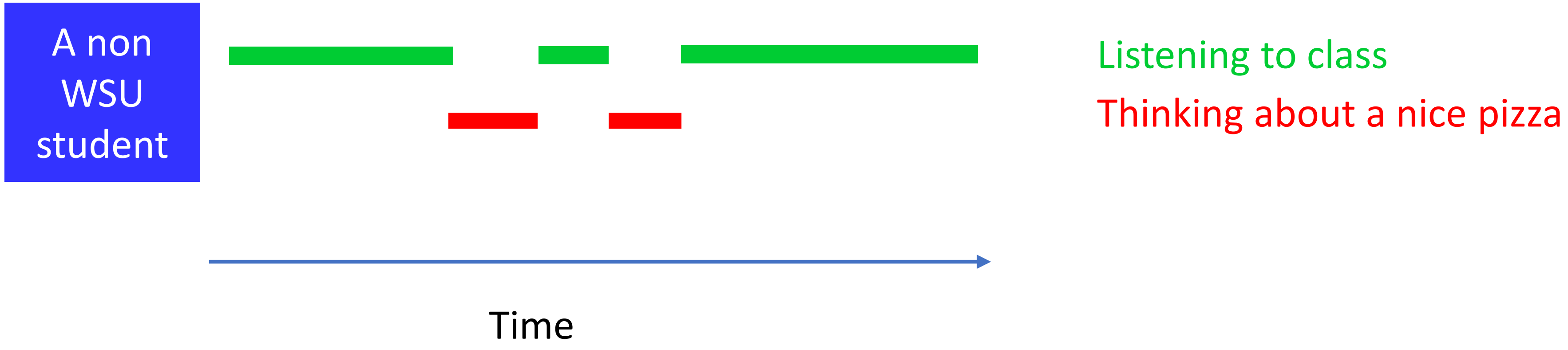
Disdvantages?

A quick revision of what you have learned

- Name all the parallel processing hardware units (starting from highest level)
  - Distributed nodes
  - Shared memory (multi-core)
  - Simultaneous multithreading (SMT)
  - Single Instruction Multiple Data (SIMD) – Vector units
  - Pipeline parallelism

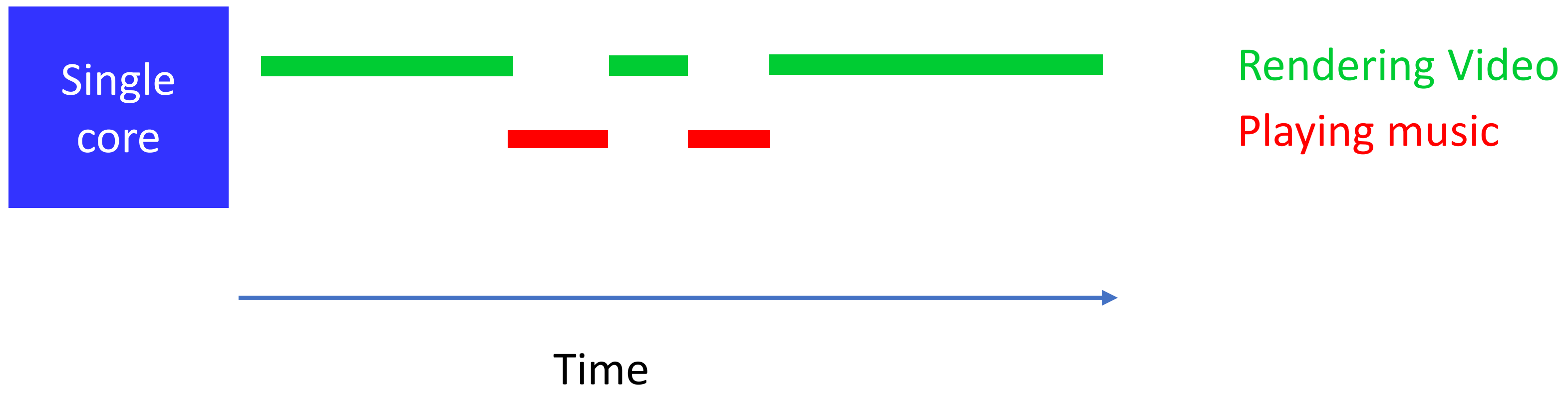
Parallelism vs Concurrency

- Concurrency: The process by which multiple jobs can be executed in an overlapped manner, one at a time



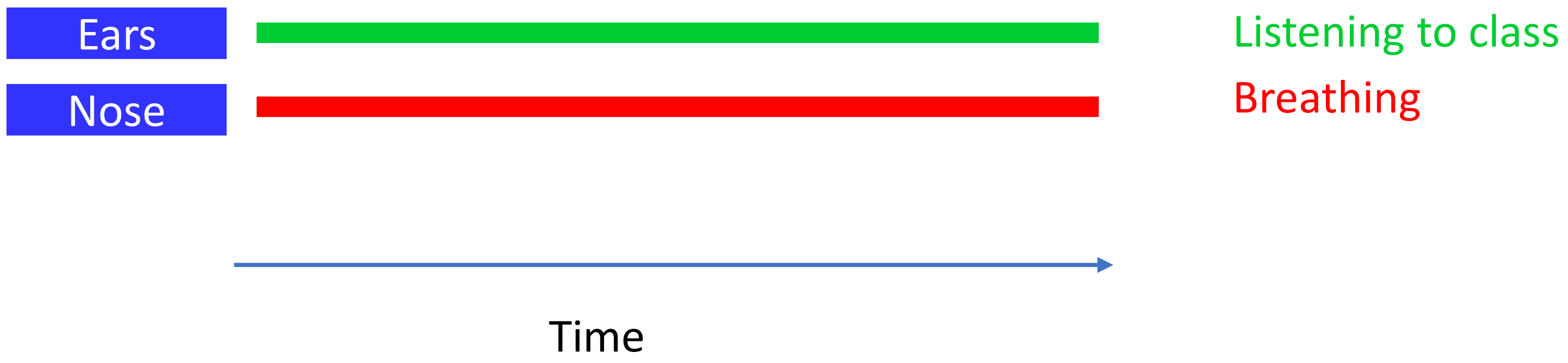
Parallelism vs Concurrency

- Concurrency: The process by which multiple jobs can be executed in an overlapped manner, one at a time



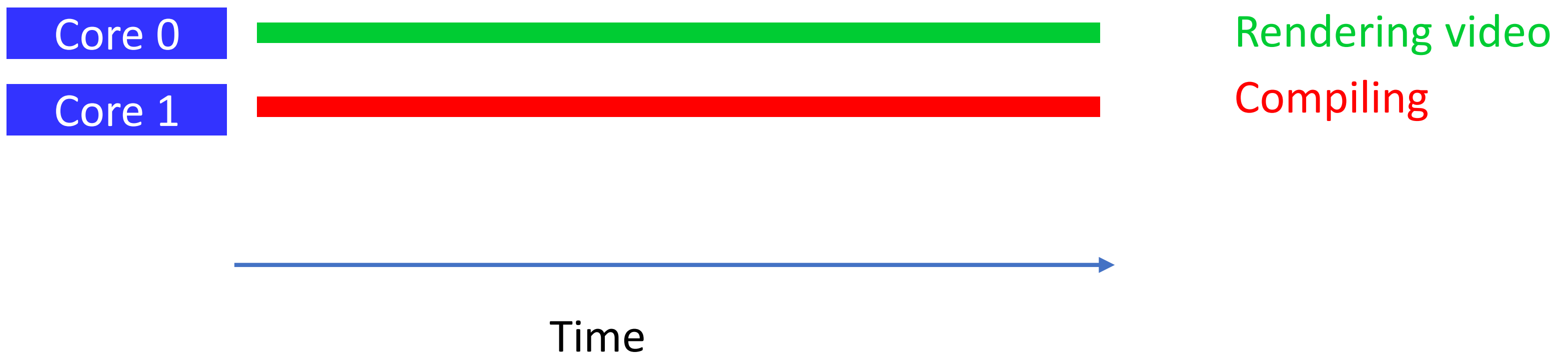
Parallelism vs Concurrency

- Parallelism: The process by which multiple jobs can be executed at the same time



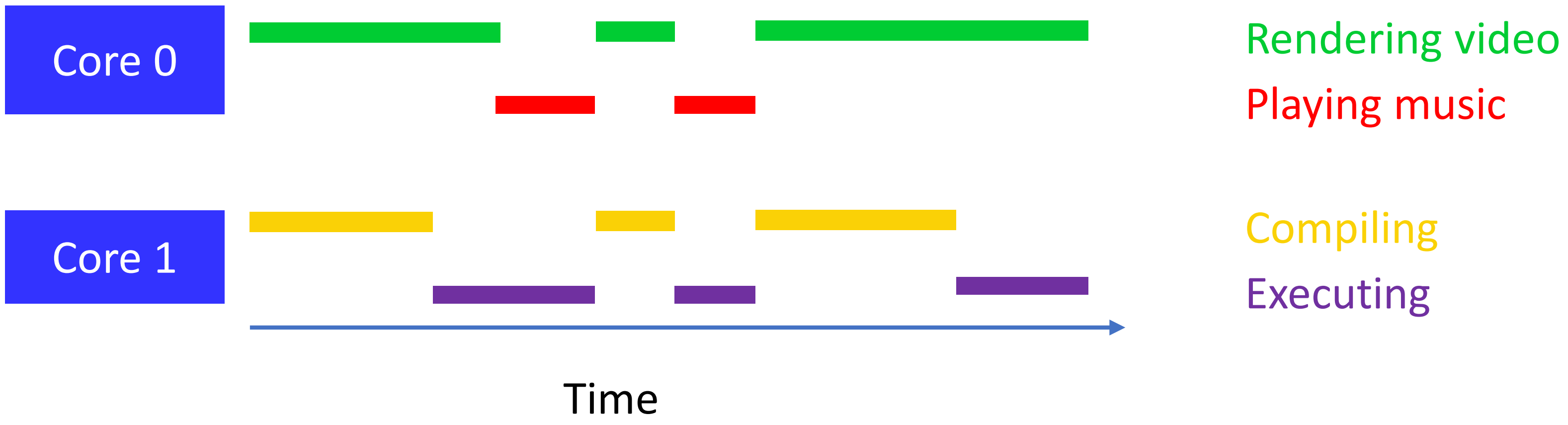
Parallelism vs Concurrency

- Parallelism: The process by which multiple jobs can be executed at the same time



Parallelism vs Concurrency

- Can we have both parallelism and concurrency?



Which of this is false?

a) An application can be concurrent and parallel
b) An application can be concurrent and not parallel
c) An application can be not concurrent and parallel
d) An application can be not concurrent and not parallel

And the answer is …. None ☺

Speedup

- What is speedup
  - SUP(P,N) = $T_{seq(N)}$ / $T_{parallel(N, P)}$
    - $T_{seq(N)}$ -- time for sequential processing
    - $T_{parallel(N, P)}$ -- time for parallel processing on P processors
    - N – problem size
  - How can we quantify?

Efficiency

- What is efficiency
  - $EFF(P,N) = T_{seq(N)} / (P * T_{parallel(N, P)}) = SUP(P,N) / P$
    - $T_{seq(N)}$ -- time for sequential processing
    - $T_{parallel(N, P)}$ -- time for parallel processing on P processors
    - N – problem size
  - What is the intuitive meaning?
    - What do you expect EFF(P,N) to be?
    - What is the best case?
      - Can it ever happen: EFF(P,N) > 1?
    - What is the worst case?
      - Can it ever happen: $T_{parallel(N, P)} > T_{seq(N)}$
  - Imagine that EFF(P,N) = 0.3
    - Is this is a good thing?
    - Is this is a bad thing?

Scalability

- What is scalability
  - The efficiency as a function of number of parallel units
- Wait wait wait
  - Efficiency???
    a) Should we measure the efficiency for a fixed problem size and a variable number of parallel units?
    b) Should we measure the efficiency by increasing both problem size and number of parallel units?

Scalability

- Measure the efficiency for a fixed problem size and a variable number of parallel units?
  - What is this question intuitively trying to answer?
  - Strong scaling (how solution time varies for a fixed problem size as we add more processors)
  - Related to Amdahl's law:
    - Let $t\_seq_{proportion}$ be the proportion of time required to execute the sequential part
    - Let $t\_par_{proportion}$ be the proportion of time required to execute the parallelizable part
    - **Strong scaling speedup = $1 / (t\_seq_{proportion} + t\_par_{proportion} / P)$**
      - What happens as P increases?
      - What does this imply?
        - For a fixed problem size, the upper bound of speedup is limited by serial part of a program.
        - Let's try to draw a graph (speedup vs P for $t\_seq_{proportion}$)

Scalability

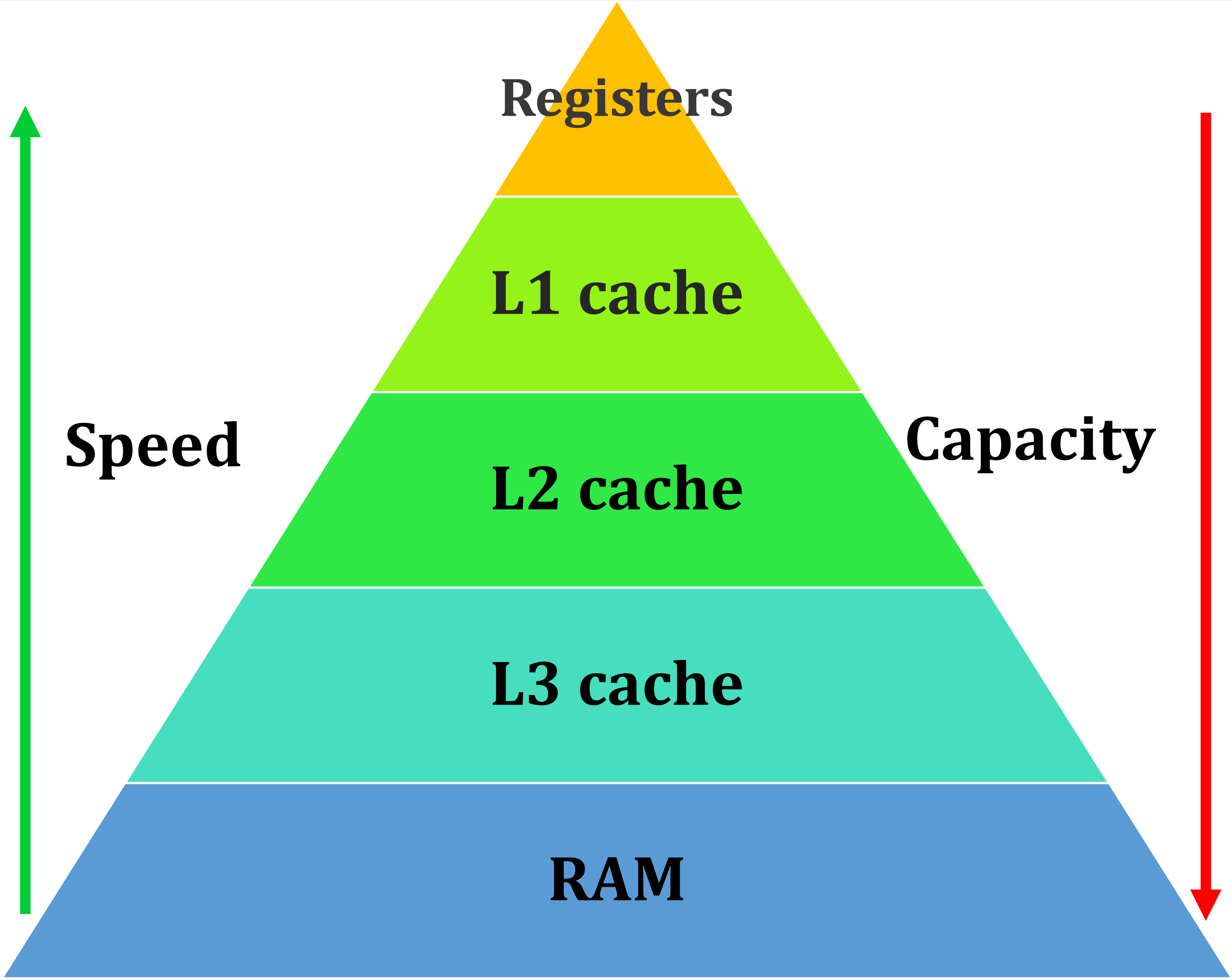
- Should we measure the efficiency by increasing both problem size and number of parallel units?
  - What is this question intuitively trying to answer?
  - Weak scaling (how the solution time varies when the problem size increases, but the work per parallel unit is kept constant)
  - Related to Gustafson's law:
    - In many cases,
      - the sequential execution time is independent of the problem size
      - The parallel part scales linearly with the number of processors (example?)
    - **In such cases the weak scaling speedup can be written as ($t\_seq_{proportion}$ + $t\_par_{proportion}$ * P)**

Scalability

- What are the factors that limit parallel scalability
  - Communication overheads
  - Startup costs
- **In general Amdahl's law is more important and difficult to achieve**

CPU Memory hierarchy

- **Registers:**
  - Capacity: ~32
  - Access time: ~1 clock cycle
- **L1 cache:**
  - Capacity: ~32KB
  - Access time: ~4-5 clock cycles
- **L2 cache:**
  - Capacity: ~256KB
  - Access time: ~10 clock cycles
- **L3 cache:**
  - Capacity: ~4 to 35MB
  - Access time: ~65 clock cycles
- **Ram:** ~300 clock cycles



CPU Memory hierarchy

- Keep the data as close to the processor as possible
  - How to do this???
    - Unfortunately formalization of this process is out of context for this course (The compiler course covers this)
    - However, we can build an intuition based on data dependencies

Data dependencies

- Two statements are said to be data-dependent if both of them accesses the same memory location at least one of them is a write operation
- Constraints the order in which the statements are scheduled

```
int a = 10;
int b = 20;
int c = a+b;
printf("%d",c);
```

Original Code

```
int b = 20;
int a = 10;
int c = a+b;
printf("%d",c);
```

Valid schedule

```
int a = 10;
int b = 20;
printf("%d",c);
int c = a+b;
```

Invalid schedule

Flow dependence

S1: a = b + c
S2: d = a + 42
S3: a=7;
S4: e = a + d

- Also called as Read-After-Write (RAW) dependence
- S_t (target statement) reads a memory that was written by S_s (source statement)
- E.g.
  - S1 -> S2 on 'a'
  - S2 -> S4 on 'd'

Anti dependence

S1: a = b + c
S2: d = a + 42
S3: a=7;
S4: e = a + d

- Also called as Write-After-Read (WAR) dependence
- S_s (source statement) reads a memory location that will be overwritten by S_t (target statement)
- E.g.
  - S2 -> S3 on ‘a’
- Can you thing about a way to avoid this dependence?

Output dependence

```
S1: a = b + c
S2: d = a + 42
S3: a=7;
S4: e = a + d
```

- Also called as Write-After-Write (WAW) dependence
- S_s (source statement) writes to a memory location that will be overwritten by S_t (target statement)
- E.g.
  - S1 -> S3 on ‘a’
- Can you thing about a way to avoid this dependence?

Input dependence

S1: a = b + c
S2: d = a + 42
S3: e = b * c

- Also called as Read-After-Read (RAR) dependence
- S_s (source statement) reads a memory location that will be read by S_t (target statement)
- E.g.
  - S1 -> S3 on ‘b’ and ‘c’
- Is this really a dependence
- Should we ever consider this?

Data dependences in Loops

```
for (int i = 1; i < 10; i++)
  A[i] = B[i-1] + C[i]      // S1
  B[i] = A[i+2] * C[i]     // S2
```

- What are all the dependences?
- The dependencies are based on dynamic instances

Data dependences in Loops

```
for (int i = 1; i < 10; i++)
  A[i] = B[i-1] + C[i]      // S1
  B[i] = A[i+2] * C[i]     // S2
```

- One easy way to see the dependences in loops is to unroll them

```
i=1  A[1] = B[0] + C[0]
     B[1] = A[3] * C[0]
i=2  A[2] = B[1] + C[1]
     B[2] = A[4] * C[1]
i=3  A[3] = B[2] + C[2]
     B[3] = A[5] * C[2]
```

A[i] -> A[i+2] //WAR(from S2 to S1)
B[i] -> B[i+1] //RAW(from S2 to S1)
C[i] -> C[i]   //RAR

Data dependences

- Two statements are said to be data-dependent if both of them accesses the same memory location at least one of them is a write operation
- Constraints the order in which the statements are scheduled
- **Why would we ever deviate from the order in which user wrote the code:**
  - **Cache friendliness**
  - **Allows parallelization**

Data dependences

- For a loop to be parallel, it should not carry any data dependences

```
for (int i = 1; i < 10; i++)
  A[i] = B[i-1] + C[i]     // S1
  B[i] = A[i+2] * C[i]     // S2
```

i loop carries dependence;
cant be parallelized

```
for (int i = 1; i < 10; i++)
  A[i] = B[i-1] + C[i]     // S1
  D[i] = E[i+2] * C[i]     // S2
```

i loop does not carry dependence;
can be parallelized

Data dependences

- For a loop to be parallel, it should not carry any data dependences

```
for (int i = 1; i < 10; i++)
   for(int j = 1; j < 10; j++)
     A[i][j] = A[i][j-1] + C[i]
```

i loop does not carry dependence;
j loop carries dependence;
i loop **can** be parallelized
j loop **cant** be parallelized

```
for (int i = 1; i < 10; i++)
   for(int j = 1; j < 10; j++)
     A[i][j] = A[i+1][j] + C[i]
```

i loop carries dependence;
j loop does not carry dependence;
i loop **cant** be parallelized
j loop **can** be parallelized

Data dependences

- For a loop to be parallel, it should not carry any data dependences
- By now, you know how to check if a loop can be parallelized or not manually. But how do you automate this?
  - Take the compiler class next semester ☺
- A lot of times, we concentrate on loop level parallelism. Why?

Data dependences

- There are loop transformations that can enable parallelism on a given loopnest (or a part of it)
  - For details take the compiler class ☺
- A simple example is given below (Loop Fission)

```
for (int i = 2; i < 10; i++)
  A[i] = B[i+1] + C[i]      // S1
  B[i] = A[i-2] * C[i]     // S2
```

i loop carries dependence;
cant be parallelized

```
for (int p = 2; p < 10; p++)
  A[p] = B[p+1] + C[p]      // S1
for (int q = 2; q < 10; q++)
  B[q] = A[q-2] * C[q]     // S2
```

p and q does not carry dependence;
Both **can** be parallelized

TA

- Pei Yau (pei-yau.weng@wsu.edu)