CS614: Advanced Compilers

Loop Parallelization

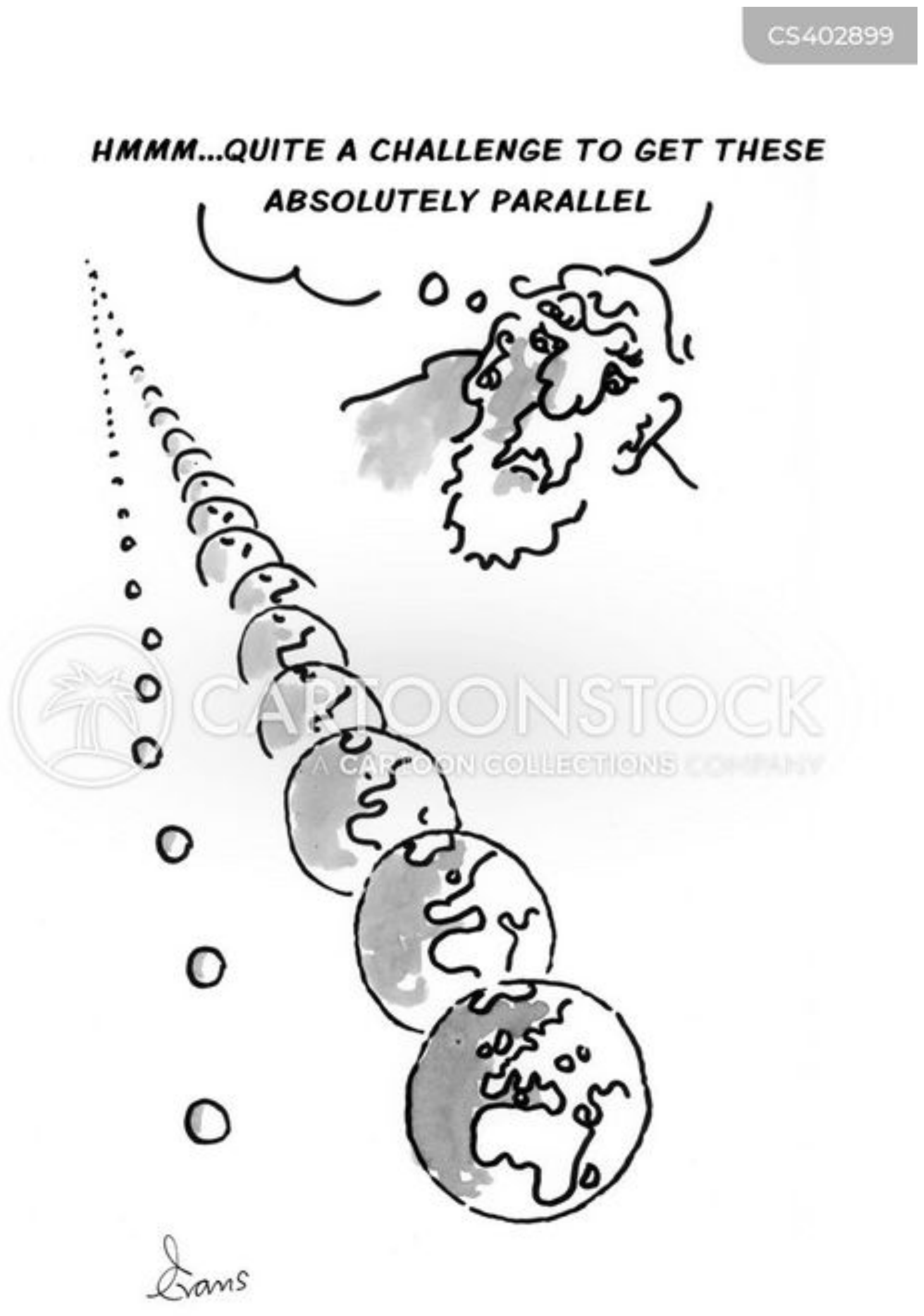
Manas Thakur
CSE, IIT Bombay

Spring 2025

Things we have learnt post midsem

- Instruction scheduling (pipelines)
- Memory optimizations (cache)
- Loop transformations (cache)

- Now lets move to compute units!
  - *We have lots of them nowadays.*
  - Utilize them effectively
  - Parallelize, vectorize



Parallelization

- What's not parallel?
  - Concurrent: Execute code sequences in an interleaved manner
- What is parallel?
  - Actually execute simultaneously (on different hardware)
- Why?
  - Moore's law is fading
  - Free lunch is over
- How?
  - Learn to code in concurrent frameworks
  - Parallelize code portions during compilation!

How much can we benefit from parallelization?

- **Amdahl's law:**
  - The speed-up achievable using parallelization is limited by the sequential portion of the code.
- For a 10-hour program, if we can parallelize 9 hours of computation, we will still take at least 1 hour.
  - That is, maximum 10x speed-up even with a 1000-core GPU!
- Thus, parallel computing with a large number of processors is straightforwardly useful only for highly parallelizable programs.
  - What's better than to target loops!!

When can't we parallelize?

- Dependencies inhibit parallelism
- Control dependency:

```
if (a < b)
   x = 10;
else
   x = 20;
```

  - Usually explicit in the syntax
  - Easier to identify
- Data dependency:
  - Quite often needs analyses to identify
  - Can `foo` and `bar` execute simultaneously?
  - Are both the `f`'s same?
  - More involved with pointers (alias analysis!)

```
foo() {
   f = 10;
}
bar() {
   print f;
}
```

```
foo() {
   x = 10;
}
bar() {
   print *y;
}
```

Kinds of data dependencies

- RAW / True / Flow:
  - `x = 2; y = x + 1;`
- WAR / Anti:
  - `y = x + 1; x = 5;`
- WAW / Output:
  - `x = 2; x = 3;`
- RAR (not critical)
  - `y = x; print x;`

True dependencies are real dependencies; anti and output dependencies can often be elided with variable/register renaming.

Kinds of loop dependencies

1. Loop-carried dependencies

- Arise because of iterations of the loop
- On every (except first) iteration, `S2` uses a value of `A` that was computed in the previous iteration by `S1`
- `S2` has a *true* dependence on `S1`, and vice-versa

```
    for (i=0; i<N; i++) {
S1:     A[i+1] = B[i];
S2:     B[i+1] = A[i];
    }
```

2. Loop-independent dependencies

- Arise because of relative statement position
- `S2` refers to the same memory location as `S1`
- There is a control-flow path from `S1` to `S2`

```
    for (i=0; i<N; i++) {
S1:     A[i] = ...;
S2:     ... = A[i];
    }
```

Which of the following loops can be parallelized?

```
for (k=0; k<n; ++k) {
    a[k] = b[k];
    b[k] = a[k] + 1;
}
```

✔

✘

```
for (k=0; k<n; ++k) {
    a[k] = a[k+1];
}
```

```
for (k=0; k<n; ++k) {
    a[x] = a[y];
}
```

?

Loop Parallelization. It is valid to convert a sequential loop to a parallel loop if the loop carries no dependence.

Array Data-Dependence Analysis

- Step 1: Express relations among array indices and loop bounds as linear inequations.
- Step 2: Determine whether solution exists using GCD test.
- Step 3: Solve after simplification if need solution values.

```
for (k=0; k<n; ++k) {
    a[x] = a[y];
}
```

```
for (i=1; i<=10; i++) {
    Z[i] = Z[i-1];
}
```

Dependence between Z[i-1] and Z[i]:

- $1 \le i_r \le 10;\ 1 \le i_w \le 10;\ i_r - 1 = i_w;\ i_r \ne i_w$
- 9 solutions: $(i_r=2, i_w=1)$, $(i_r=3, i_w=2)$, ...

Dependence between Z[i] and itself:

- $1 \le i_{w1} \le 10;\ 1 \le i_{w2} \le 10;\ i_{w1} = i_{w2};\ i_{w1} \ne i_{w2}$
- No solution

Array Data-Dependence Analysis (Cont.)

GCD Test. The linear *Diophantine* equation

$$a_1x_1 + a_2x_2 + \ldots + a_nx_n = c$$

has an integer solution for $x_1, x_2, \ldots, x_n$ if and only if $\gcd(a_1, a_2, \ldots, a_n)$ divides c.

```
for (i = 1; i < 10; i++) {
   Z[2*i] = 10;
}
for (j = 1; j < 10; j++) {
   Z[2*j+1] = 20;
}
```

- In order for the two `Z` accesses to write to the same location, `2*i = 2*j+1`.
- Canonical form: `2*i - 2*j = 1`.
- `GCD(2,-2) = 2`, which does not divide `1`.
- Thus there is no dependency.
- Hence the two loops can run in parallel.

Loop Parallelization: Practice

```
for (i = 1; i <= 4; i++) {
   B[i] = A[3*i-5] + 2.0;
   A[2*i+1] = 1.0/i;
}
```

- $2*i_1+1 = 3*i_2-5$
- $2*i_1 - 3*i_2 = -6$
- GCD(2,-3) = 1, which divides -6
- Hence the loop cannot be parallelized.
- a[7] is written to in iteration 4 and used in iteration 3.

```
for (i = 1; i <= 4; i++) {
   B[i] = A[4*i] + 2.0;
   A[2*i+1] = 1.0/i;
}
```

- $2*i_1+1 = 4*i_2$
- $2*i_1 - 4*i_2 = -1$
- GCD(2,-4) = 2, which does not divide -1
- Hence the loop can be parallelized.

Reordering within a loop

- A less obvious loop-independent dependence:

```
   for (i=1; i<9; i++) {
S1:   A[i] = ...;
S2:   ... = A[10-i];
   }
```

- Dependence in 5th iteration.

Why this requirement?

If there is a loop-independent dependence from S1 to S2, any reordering transformation that does not move statement instances across iterations and preserves the relative order of S1 and S2 in the loop body preserves that dependence.

Reordering within a loop (Cont.)

- Some compiler optimization may perform this transformation:

```
   for (i=1; i<N; i++) {
S1:    A[i] = B[i] + C;
S2:    D[i] = A[i] + E;
   }
```

→

```
   D[1] = A[1] + E;
   for (i=2; i<N; i++) {
S1:    A[i-1] = B[i-1] + C;
S2:    D[i] = A[i] + E;
   }
   A[N] = B[N] + C;
```

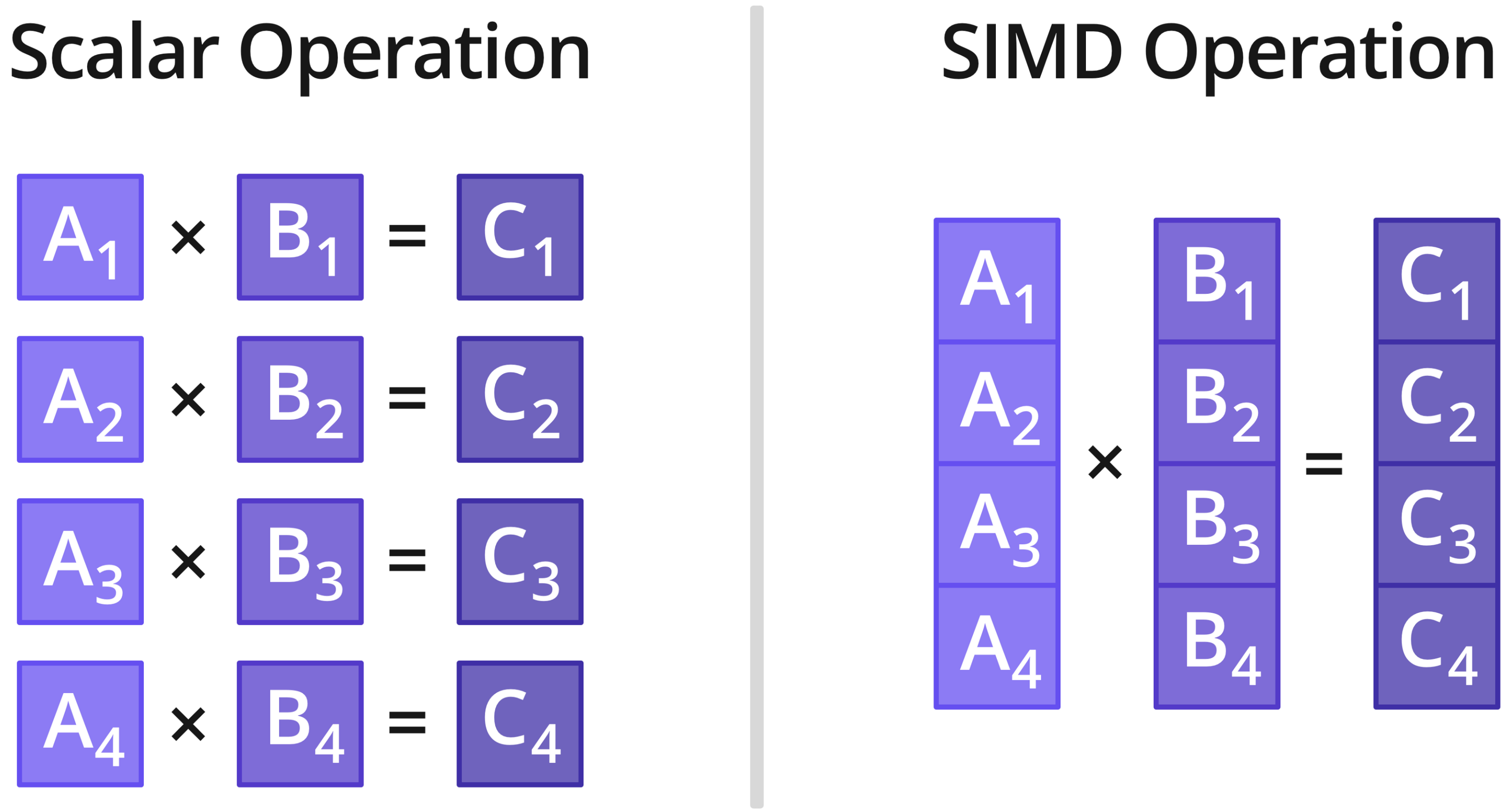
- Order of statements within the body preserved, but loop-independent true dependence converted to anti-dependence ==> *invalid transformation*.

> A loop-carried dependence is satisfied so long as loops are iterated in the original order, regardless of the statement order within a specific iteration. A loop-independent dependence is satisfied so long as the statement order is maintained, regardless of the order in which the loops are iterated.

Loop Vectorization

Vector Operation

- An operation that can be performed simultaneously on multiple registers
- SIMD (Single Instruction, Multiple Data) processors



- Most modern hardware support vector instructions (sometimes, of varied lengths)
  - Intel has AVX; AMD has 3DNow!; ARM has NEON; GPUs.
  - Some also support "predicated" (masked) vectorization.

Compiling for Vector Pipelines

- Some programming languages provide capability for manual vectorization
  - `#pragma omp simd` in OpenMP; `_mm_add_ps` in C
  - Java? Programmer shouldn't bother with tricky stuff; offload to compiler ;-)
- Many compilers perform auto-vectorization as a machine-dependent optimization

```
for (i=0; i<1024; i++) {
   C[i] = A[i] * B[i];
}
```

```
for (i=0; i<1024; i+=4) {
   C[i:i+3] = A[i:i+3] * B[i:i+3];
}// remaining array, if needed
```

- How is this different from loop unrolling?
  - *The four multiplications must have the effect of getting performed in parallel.*

```
for (i=0; i<1024; i+=4) {
   C[i] = A[i] * B[i];
   C[i+1] = A[i+1] * B[i+1];
   C[i+2] = A[i+2] * B[i+2];
   C[i+3] = A[i+3] * B[i+3];
}
```

Vectorization

- Valid transformation:

```
for (i=1; i<N; i++) {
   X[i] = X[i] + C;
}
```

→

```
X[1:N] = X[1:N] + C
```

- Invalid transformation:

```
for (i=1; i<N; i++) {
   X[i+1] = X[i] + C;
}
```

→

```
X[2:N+1] = X[1:N] + C
```

- Sequential version uses a value of X that is computed in the previous iteration, while the vectorized version uses old values of X.

Which loops can be vectorized?

```
   for (i=1; i<N; i++) {
S1:    A[i+1] = B[i] + C;
S2:    D[i] = A[i] + E;
   }
```

→

```
S1: A[2:N+1] = B[1:N] + C
S2: D[1:N] = A[1:N] + E
```

- Valid vectorization even though the loop carries a dependence!
- We had an implicit *loop distribution/fission*:

```
   for (i=1; i<N; i++) {
S1:    A[i+1] = B[i] + C;
S2:    D[i] = A[i] + E;
   }
```

→

```
   for (i=1; i<N; i++) {
S1:    A[i+1] = B[i] + C;
   }
   for (i=1; i<N; i++) {
S2:    D[i] = A[i] + E;
   }
```

- Did you say it worked because the loop-carried dependence was *forward*?

Which loops can be vectorized? (Cont.)

- The fission magic can sometimes be made to work even when the loop-carried dependence is *backward*, with reordering within the loop body:

```
   for (i=1; i<N; i++) {
S2:   D[i] = A[i] + E;
S1:   A[i+1] = B[i] + C;
   }
```

→

```
   for (i=1; i<N; i++) {
S1:   A[i+1] = B[i] + C;
S2:   D[i] = A[i] + E;
   }
```

- However, this case has a backward-carried dependence as well as a loop-independent dependence, and hence interchange will not work:

```
   for (i=1; i<N; i++) {
S1:   B[i] = A[i] + E;
S2:   A[i+1] = B[i] + C;
   }
```

Loop Vectorization. A statement contained in at least one loop can be vectorized by direct rewriting if the statement is not included in any cycle of dependences.

Which reorderings are valid compiler transformations?

- We have studied (at least) the following reordering transformations:
  - ***Software pipelining; data prefetching; LICM; loop interchange, tiling, fusion, fission, parallelization, vectorization.***
- Any general notion that describes which ones are valid?
- Interestingly, validity depends (quite a bit) on the programming language!
- The **memory model** for a language L determines how much "reordering freedom" do compiler implementations and hardware have for programs written in L.
- Decided by PL designers (often continuously improvised by PL researchers).

Next Class: *Memory Models* and Valid Reordering Transformations.