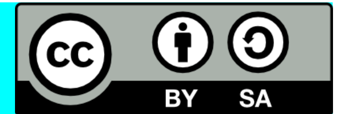
Parallelizing Loops

Moreno Marzolla
Dip. di Informatica—Scienza e Ingegneria (DISI)
Università di Bologna

moreno.marzolla@unibo.it

Copyright © 2017–2023
Moreno Marzolla, Università di Bologna, Italy
https://www.moreno.marzolla.name/teaching/HPC/



This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License (CC BY-SA 4.0). To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/ or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

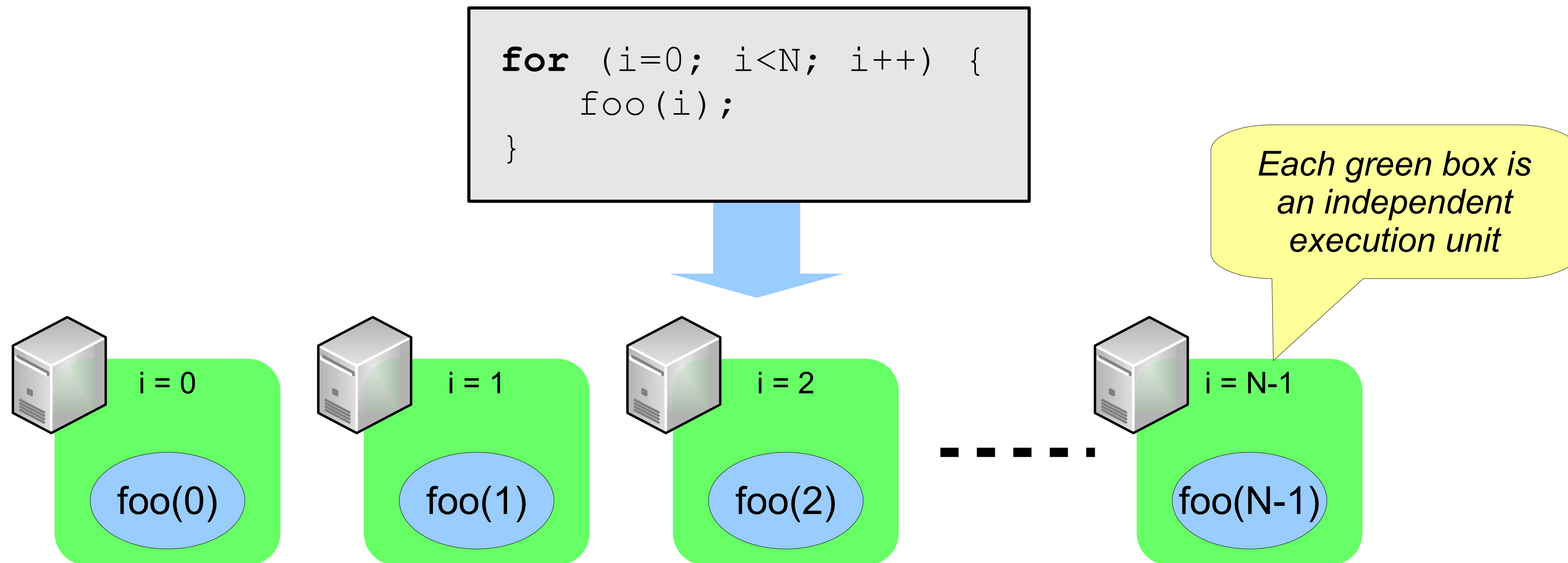
Credits

- Salvatore Orlando (Univ. Ca' Foscari di Venezia)
- Mary Hall (Univ. of Utah)

Loop optimization

- 90% of execution time in 10% of the code
  - Mostly in loops
- Loop optimizations
  - Transform loops preserving the semantics
- Goal
  - Single-threaded system: mostly optimizing for memory hierarchy
  - Multi-threaded and vector systems: loop parallelization

Executing loops in parallel

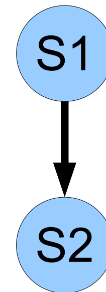


- Loop iterations are executed in parallel
  - The number of execution units might be much less than the number of loop iterations
  - therefore, each exec. unit takes care of multiple loop iterations in some unspecified order

Data Dependence

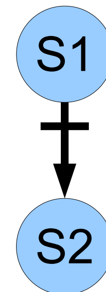
- Two memory accesses are involved in a data dependence if they may refer to the same memory location and one of the references is a write
- Data-Flow or true dependence
  - RAW (Read After Write)

```
S1  a = b + c;
S2  d = 2 * a;
```



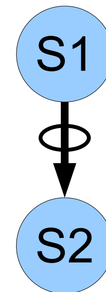
- Anti dependence
  - WAR (Write After Read)

```
S1  c = a + b;
S2  a = 2 * a;
```



- Output dependence
  - WAW (Write After Write)

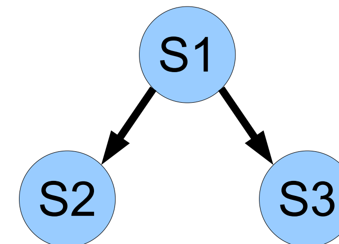
```
S1  a = k;
    if (a>0) {
S2      a = 2 * c;
    }
```



Control Dependence

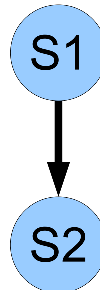
- An instruction S2 has a control dependence on S1 if the outcome of S1 determines whether S2 is executed or not
  - Of course, S1 and S2 can not be exchanged
- This type of dependence applies to the condition of an if-then-else or loop with respect to their bodies

```
S1  if (a>0) {
S2      a = 2 * c;
    } else {
S3      b = 3;
    }
```



Dependence

- In the following, we always use a simple arrow to denote any type of dependence
  - S2 depends on S1

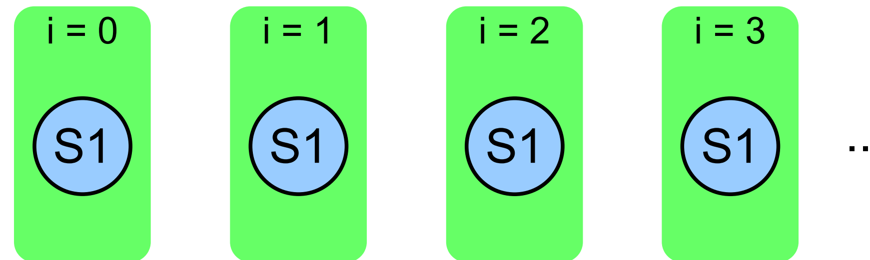


Fundamental Theorem of Dependence

- *“Any reordering transformation that preserves every dependence in a program preserves the meaning of that program”*
- Recognizing parallel loops (intuitively)
  - Find data dependencies in loop
  - No dependencies crossing iteration boundary → loop can be parallelized

Example

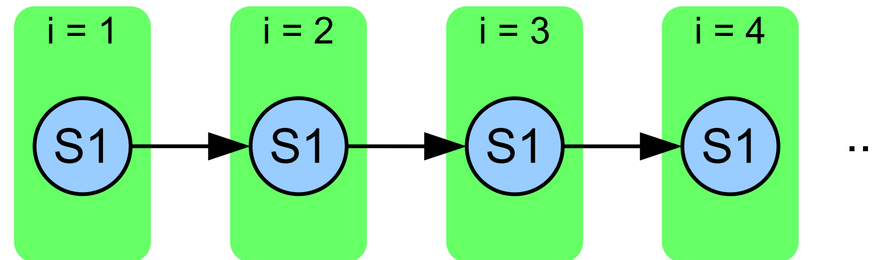
```
for (i=0; i<n; i++) {
S1 a[i] = b[i] + c[i];
}
```



- Each iteration does not depend on previous ones
  - There are no dependences crossing iteration boundaries
- This loop is fully parallelizable
  - Loop iterations can be performed concurrently, in any order
  - Iterations can be split across processors

Example

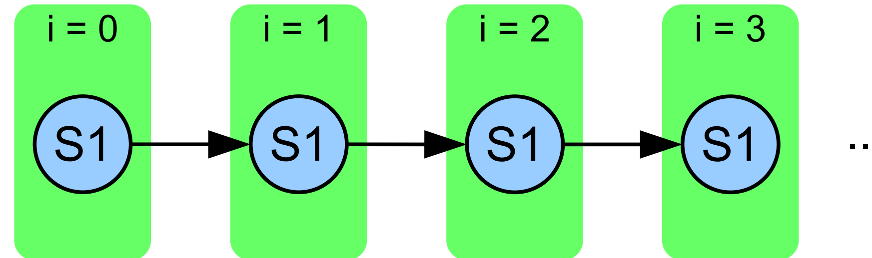
```
for (i=1; i<n; i++) {
S1 a[i] = a[i-1] + b[i]
}
```



- Each iteration depends on the previous one (RAW)
  - Loop-carried dependence
- Hence, this loop is not parallelizable with trivial transformations

Example

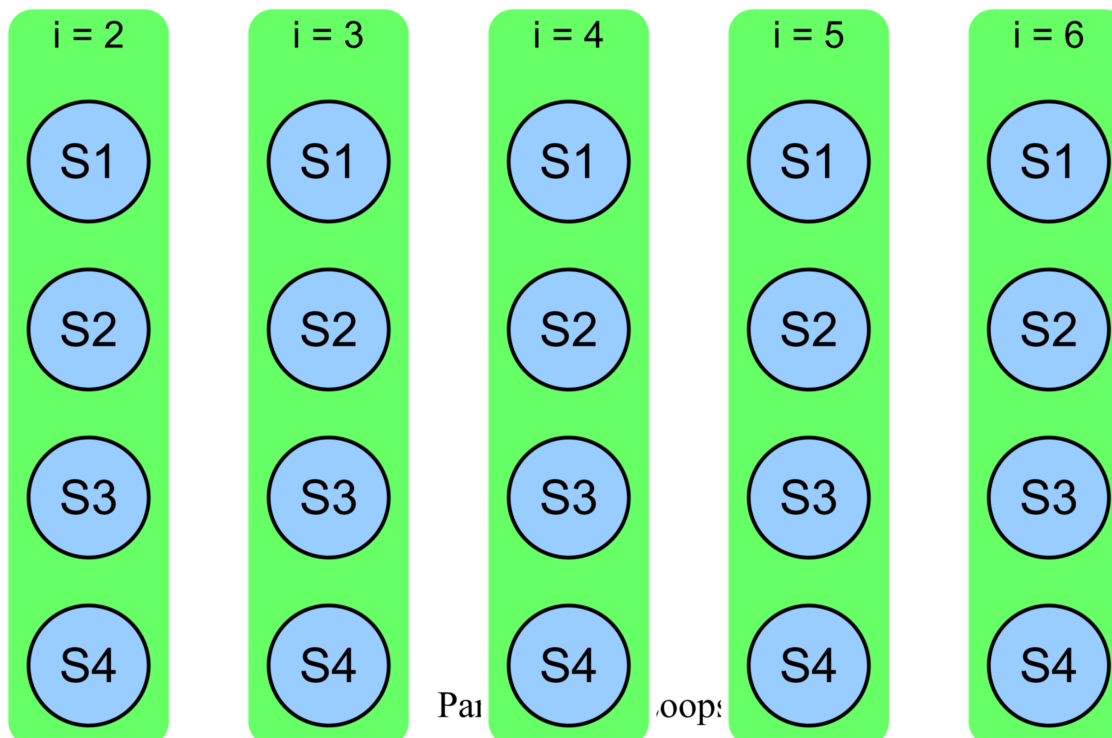
```
s = 0;
for (i=0; i<n; i++) {
S1  s = s + a[i];
}
```



- We have a loop-carried dependence on s that can not be removed with *trivial* loop transformations
  - but can be removed with *non-trivial* transformations
  - this is a reduction, indeed!

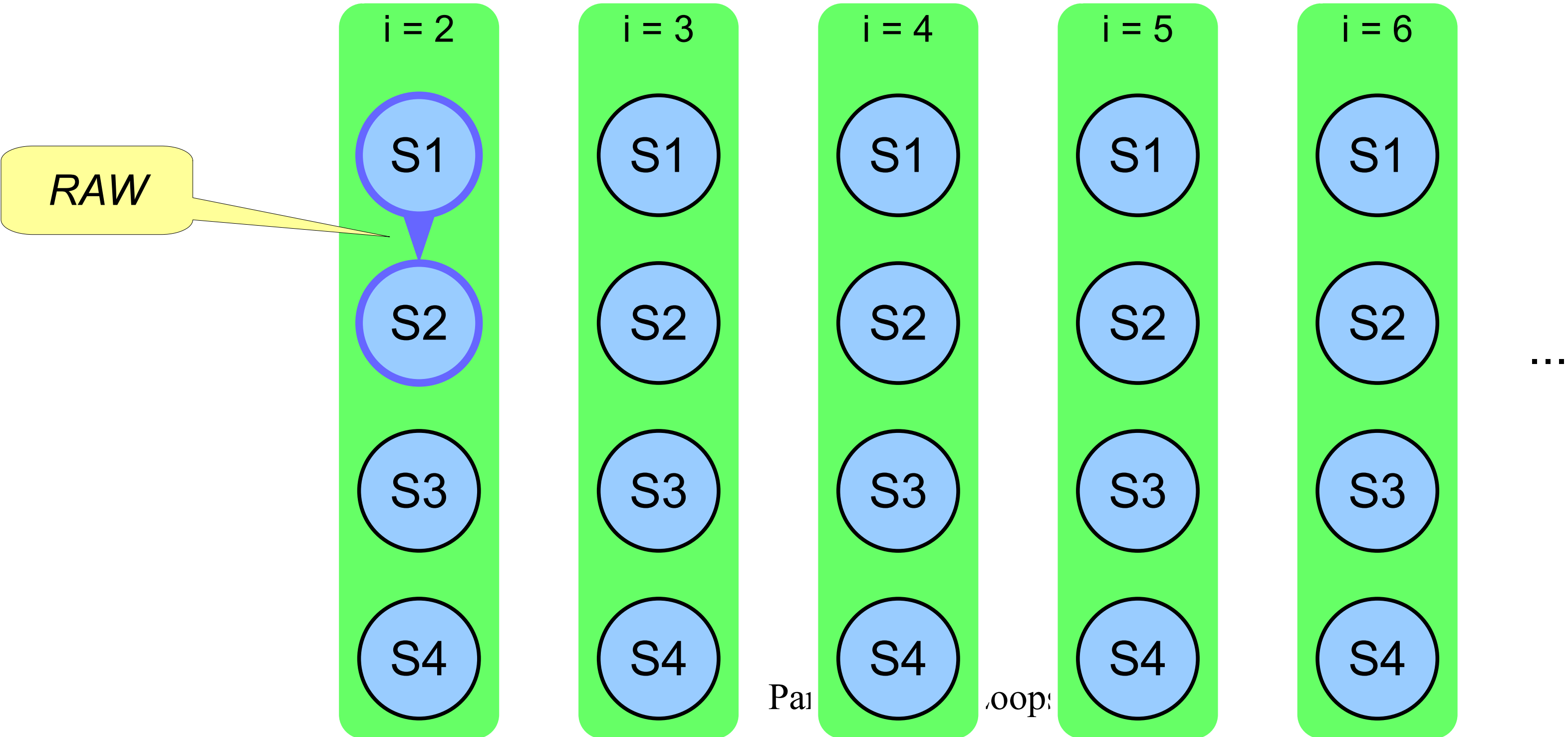
Exercise: draw the dependences among iterations of the following loop

```
for (i=2; i<n; i++) {
S1  a[i] = 4 * c[i-1] - 2;
S2  b[i] = a[i] * 2;
S3  c[i] = a[i-1] + 3;
S4  d[i] = b[i] + c[i-2];
}
```



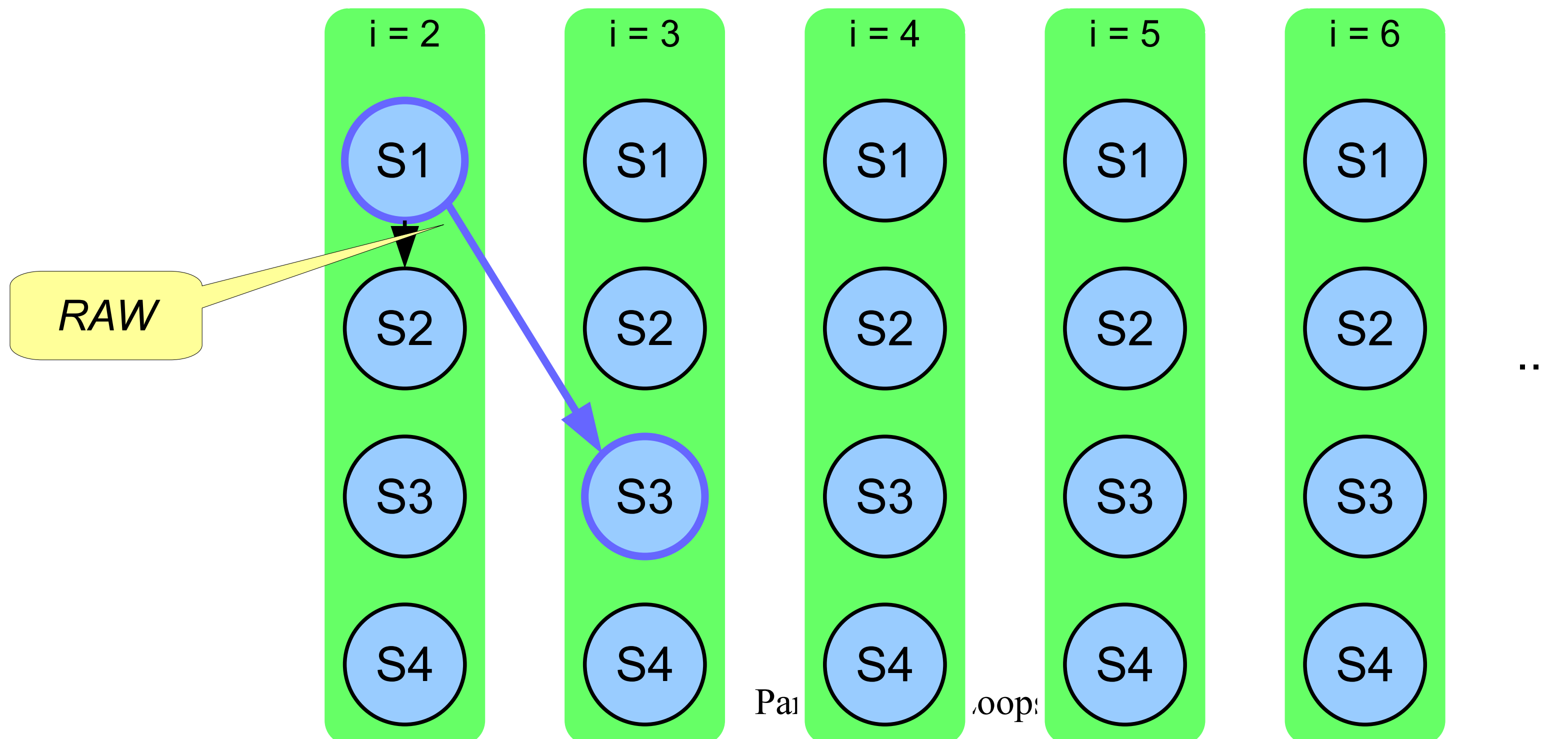
Exercise (cont.)

```
for (i=2; i<n; i++) {
S1  a[i] = 4 * c[i-1] - 2;
S2  b[i] = a[i] * 2;
S3  c[i] = a[i-1] + 3;
S4  d[i] = b[i] + c[i-2];
}
```



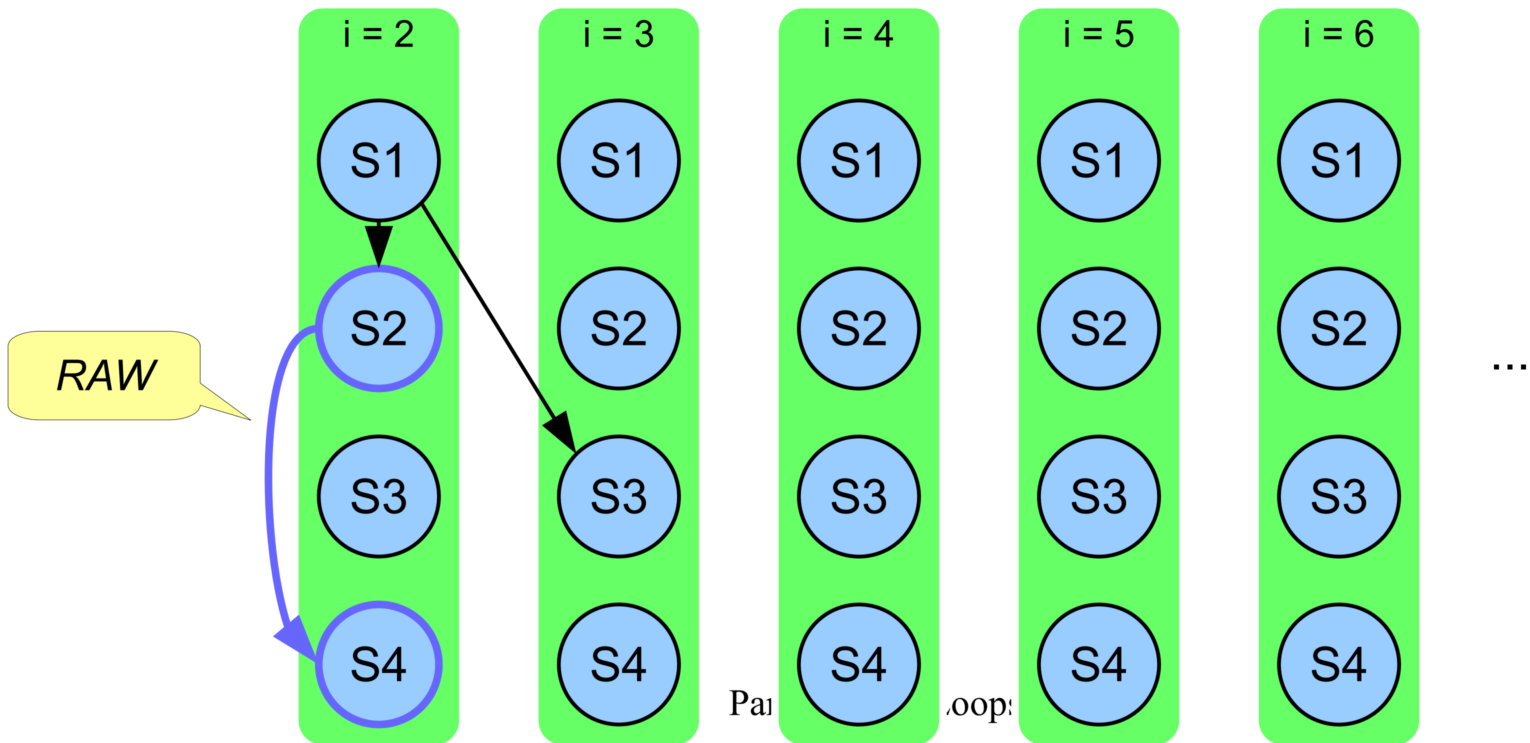
Exercise (cont.)

```
for (i=2; i<n; i++) {
S1  a[i] = 4 * c[i-1] - 2;
S2  b[i] = a[i] * 2;
S3  c[i] = a[i-1] + 3;
S4  d[i] = b[i] + c[i-2];
}
```



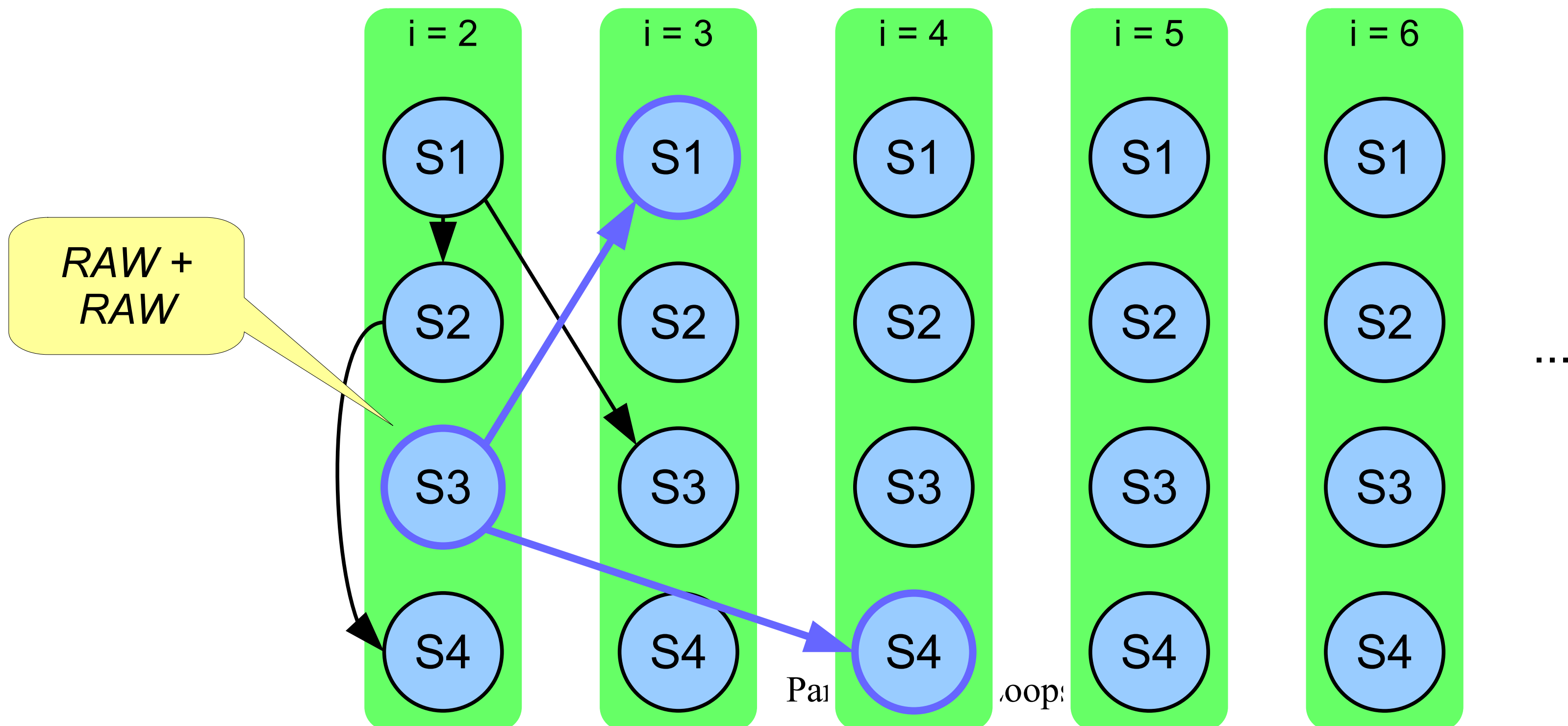
Exercise (cont.)

```
for (i=2; i<n; i++) {
S1   a[i] = 4 * c[i-1] - 2;
S2   b[i] = a[i] * 2;
S3   c[i] = a[i-1] + 3;
S4   d[i] = b[i] + c[i-2];
}
```



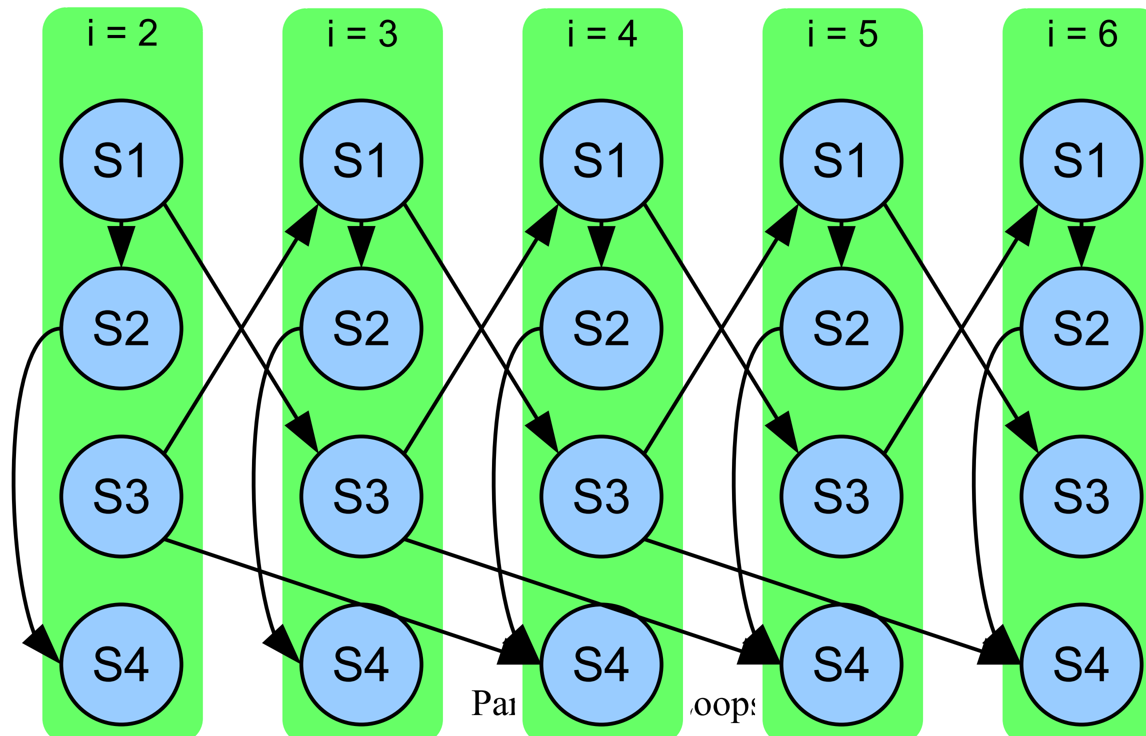
Exercise (cont.)

```
for (i=2; i<n; i++) {
S1  a[i] = 4 * c[i-1] - 2;
S2  b[i] = a[i] * 2;
S3  c[i] = a[i-1] + 3;
S4  d[i] = b[i] + c[i-2];
}
```



Exercise (cont.)

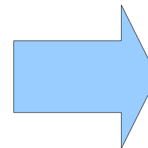
```
for (i=2; i<n; i++) {
S1  a[i] = 4 * c[i-1] - 2;
S2  b[i] = a[i] * 2;
S3  c[i] = a[i-1] + 3;
S4  d[i] = b[i] + c[i-2];
}
```



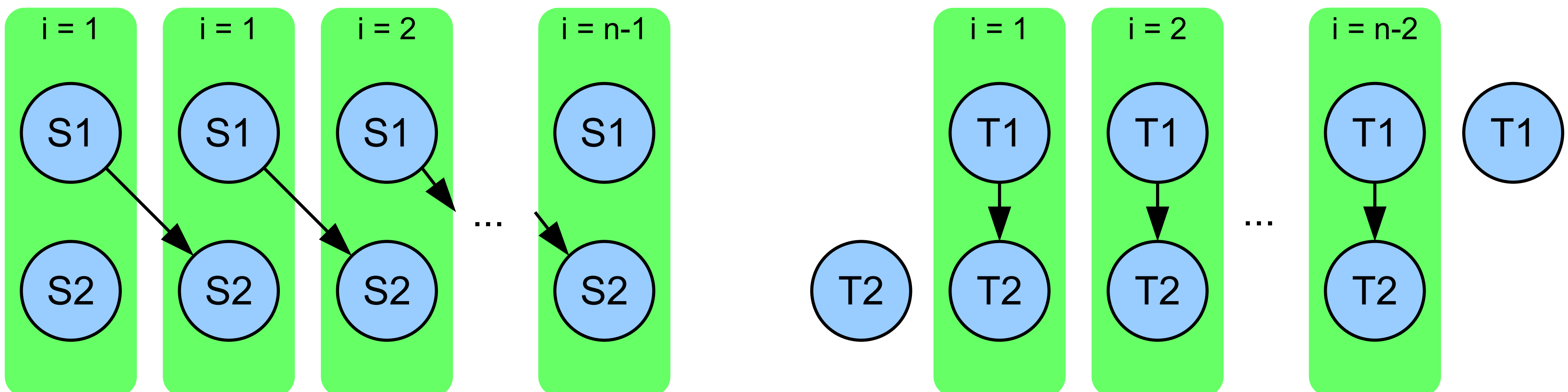
Removing dependences: Loop aligning

- Dependences can sometimes be removed by aligning loop iterations

```
a[0] = 0;
for (i=1; i<n; i++) {
S1 a[i] = b[i-1] * c[i];
S2 d[i] = a[i-1] + 2;
}
```



```
a[0] = 0;
d[1] = a[0] + 2;
for (i=1; i<n-1; i++) {
T1 a[i] = b[i-1] * c[i];
T2 d[i+1] = a[i] + 2;
}
a[n-1] = b[n-2] * c[n-1];
```



Loop interchange

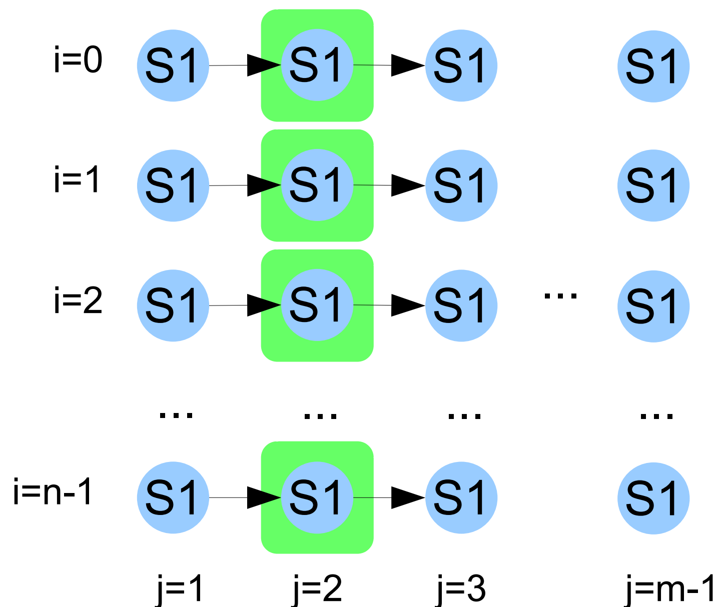
- Exchanging the loop indexes might allow the outer loop to be parallelized
  - Why? To use coarse-grained parallelism (if appropriate)

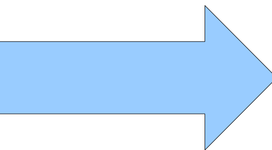
```
for (j=1; j<m; j++) {
  for (i=0; i<n; i++) {
S1  a[i][j] = a[i][j-1] + b[i];
  }
}
```

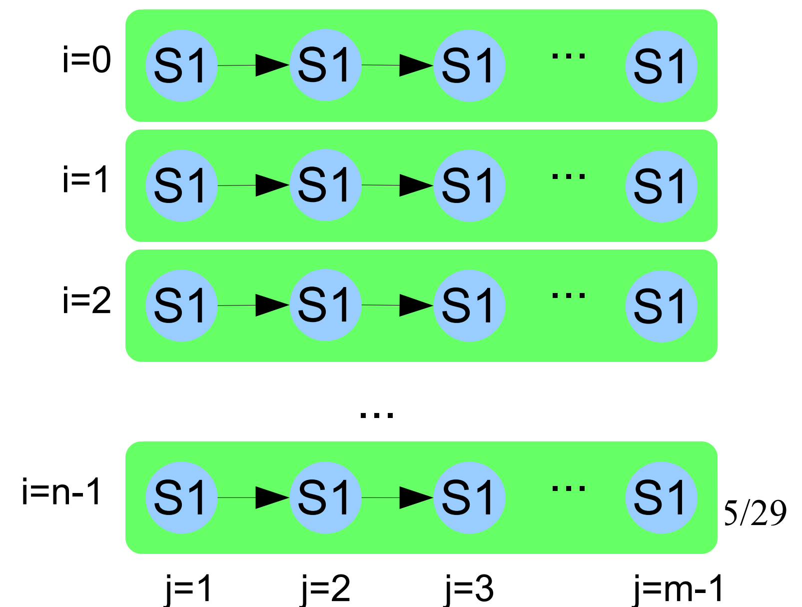
```
for (i=0; i<n; i++) {
  for (j=1; j<m; j++) {
S1  a[i][j] = a[i][j-1] + b[i];
  }
}
```

Parallelize the inner loop

Parallelize the outer loop

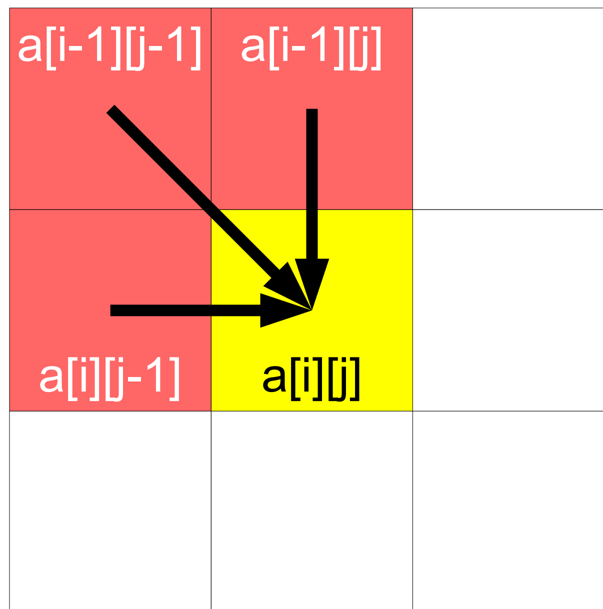


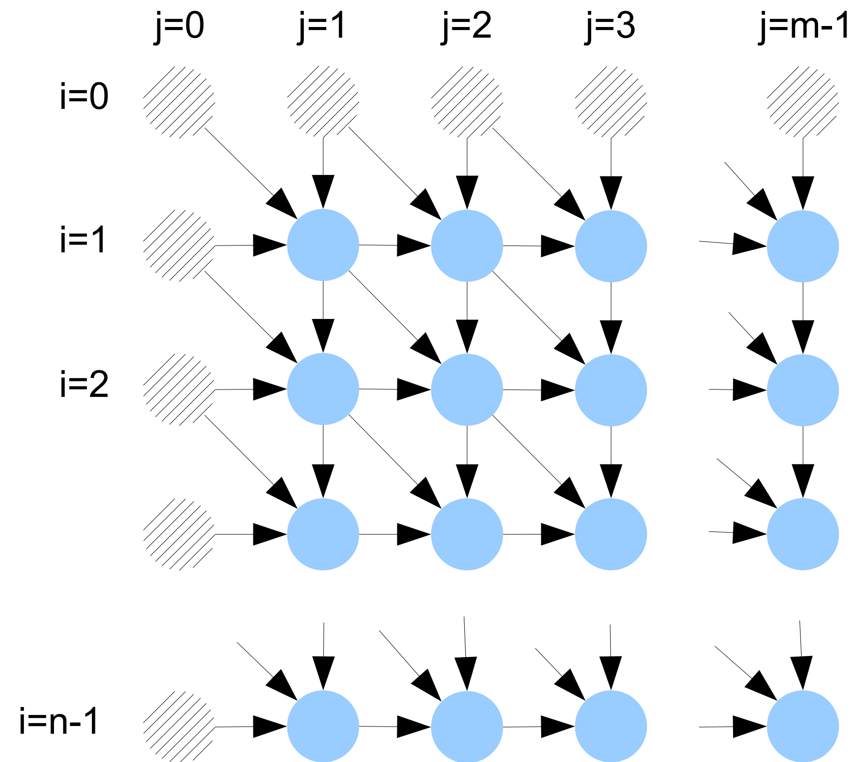




Difficult dependences

```
for (i=1; i<n; i++) {
  for (j=1; j<m; j++) {
    a[i][j] = a[i-1][j-1] +
              a[i-1][j  ] +
              a[i  ][j-1];
  }
}
```

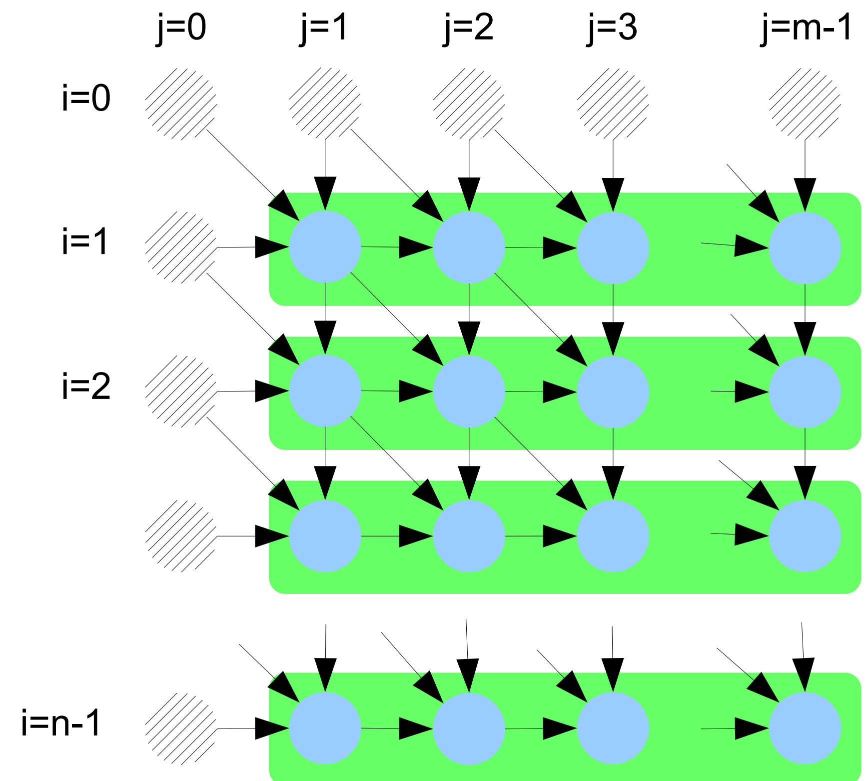




Difficult dependences

- It is not possible to parallelize the loop iterations no matter what loop we consider

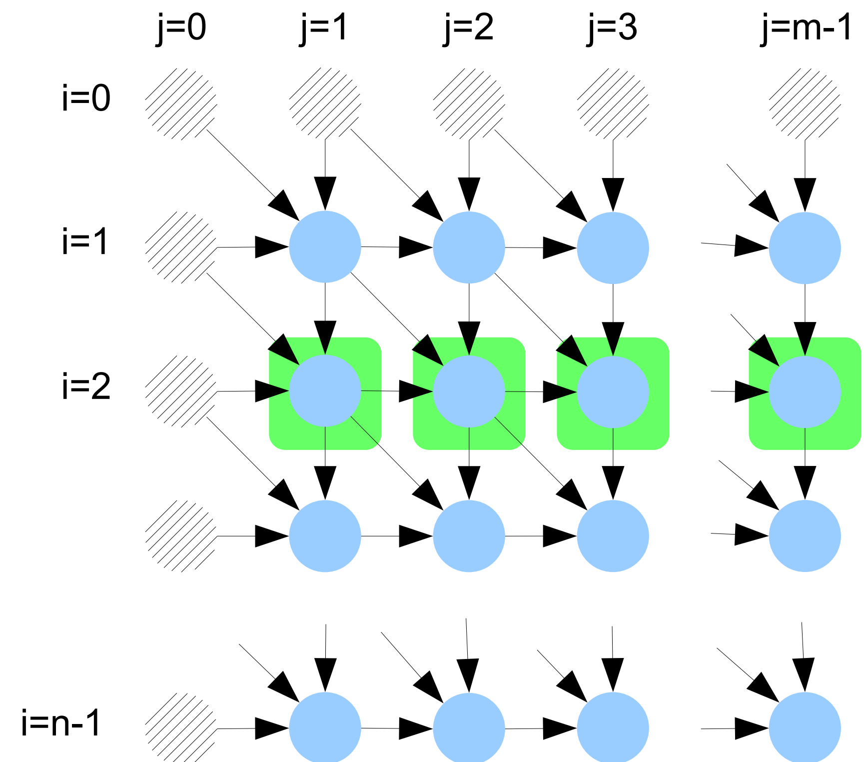
```
for (i=1; i<n; i++) {
  for (j=1; j<m; j++) {
    a[i][j] = a[i-1][j-1] +
              a[i-1][j  ] +
              a[i  ][j-1];
  }
}
```



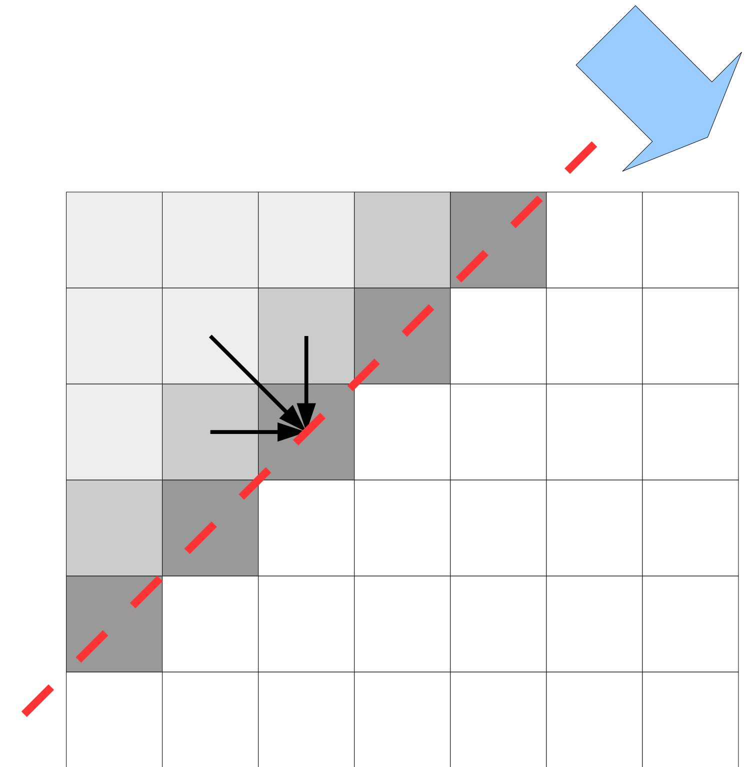
Difficult dependences

- It is not possible to parallelize the loop iterations no matter what loop we consider

```
for (i=1; i<n; i++) {
  for (j=1; j<m; j++) {
    a[i][j] = a[i-1][j-1] +
              a[i-1][j  ] +
              a[i  ][j-1];
  }
}
```



Solution



- It is possible to parallelize the inner loop by sweeping the matrix diagonally
  - *Wavefront sweep*

```
for (slice=0; slice < n + m - 1; slice++) {
    z1 = slice < m ? 0 : slice - m + 1;
    z2 = slice < n ? 0 : slice - n + 1;
    /* The following loop can be parallelized */
    for (i = slice - z2; i >= z1; i--) {
        j = slice - i;
        /* process a[i][j] … */
    }
}
```


https://stackoverflow.com/questions/2112832/traverse-rectangular-matrix-in-diagonal-strips

Some exercises
(more during the labs)

Exercise

- Can this loop be parallelized? How?

```
#define N some_big_number
double a[N];
const double f = 1.000001;
double val = 1.0;
int i;

for (i=0; i<N; i++) {
    a[i] = val;
    val = val * f
}
```

Exercise

- Which loop(s) (if any) can be parallelized, and why?

```
#define N some_big_number
int s[N] = {1, 1, ... 1};
double p[N] = { ... };
int i, j;

/* Assume that all s[i] are initially 1, and that all
p[i] have been suitably initialized. Assume that
function f() has no side effects */

for (i=0; i<N; i++) {
    if (s[i]) {
        for (j=0; j<N; j++) {
            if (s[j] && f(p[i], p[j])) {
                s[j] = 0;
            }
        }
    }
}
```

Exercise

- Which loop(s) can be parallelized? Why?

```
#define N 10000
double phi[2][N][N], maxdelta;
const double EPS = 1.0e-6;
int cur = 0, next = 1;

/* ...Initializations not shown... */

do {
  maxdelta = 0.0;
  for (int i=1; i<N-1; i++) {
    for (int j=1; j<N-1; j++) {
      phi[next][i][j] = (phi[cur][i+1][j] + phi[cur][i-1][j] +
                         phi[cur][i][j+1] + phi[cur][i][j-1]) / 4;
      const double delta = fabs(phi[next][i][j] - phi[cur][i][j]);
      if (delta > maxdelta) {
        maxdelta = delta;
      }
    }
  }
   /* exchange “cur” and “next” */
  const int tmp = cur; cur = next; next = tmp;
} while (maxdelta > EPS);
```

Conclusions

- A loop can be parallelized if there are no dependences crossing loop iterations
- Some kinds of dependences can be eliminated by
  - applying code transformations (*reordering*, *aligning*)
  - using patterns (e.g., *scan*, *reduction*)
  - sweeping across iterations "diagonally"
- There are cases where dependences can simply not be removed