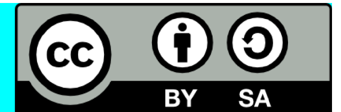
Parallelizing Loops

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

http://www.moreno.marzolla.name/

Copyright © 2017
Moreno Marzolla, Università di Bologna, Italy
(http://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

Reordering instructions

- When can you change the order of two instructions without changing the semantics?
  - They do not operate (read or write) on the same variables
  - They can be only read the same variables
- This is formally captured in the concept of data dependence
  - True dependence: Write X – Read X (RAW)
  - Output dependence: Write X – Write X (WAW)
  - Anti dependence: Read X – Write X (WAR)
- If you detect Read X – Read X (RAR) it is safe to change the order

Race Condition vs Data Dependence

- A race condition exists when the result of an execution depends on the timing of two or more events
- A data dependence is an ordering on a pair of memory operations that must be preserved to maintain correctness

Key Control Concept: Data Dependence

- Q: When is parallelization guaranteed to be safe?
- A: If there are no data dependencies across reordered computations.
- Definition: Two memory accesses are involved in a data dependence if they may refer to the same memory location and one of the accesses is a write.
- Bernstein's conditions
  - I_j is the set of memory locations read by process P_j
  - O_j the set of memory locations written by P_j
  - To execute P_j and another process P_k in parallel, the following conditions must hold

$I_j \cap O_k = \varnothing$ write after read

$I_k \cap O_j = \varnothing$ read after write

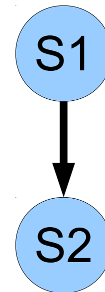
$O_j \cap O_k = \varnothing$ write after write

Bernstein, A. J. (1 October 1966). "Analysis of Programs for Parallel Processing". IEEE Transactions on Electronic Computers. EC-15 (5): 757–763. doi:10.1109/PGEC.1966.264565

Data Dependence

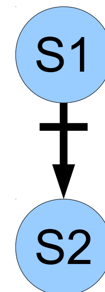
- Two memory accesses are involved in a data dependence if they may refer to the same memory location and one of the references is a write
- Data-Flow or true dependency
  - RAW (Read After Write)

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



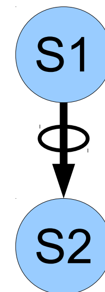
- Anti dependency
  - WAR (Write After Read)

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



- Output dependency
  - WAW (Write After Write)

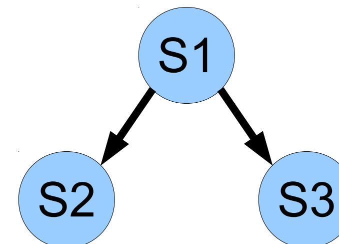
```
S1  a = k;
    if (a>0) {
S2      a = 2 * c;
    }
```



Control Dependence

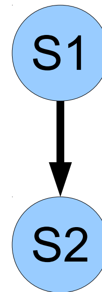
- An instruction S2 has a control dependency on S1 if the outcome of S1 determines whether S2 is executed or not
  - Of course, S1 and S2 can not be exchanged
- This type of dependency applies to the condition of an if-then-else or loop with respect to their bodies

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



Data Dependence

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

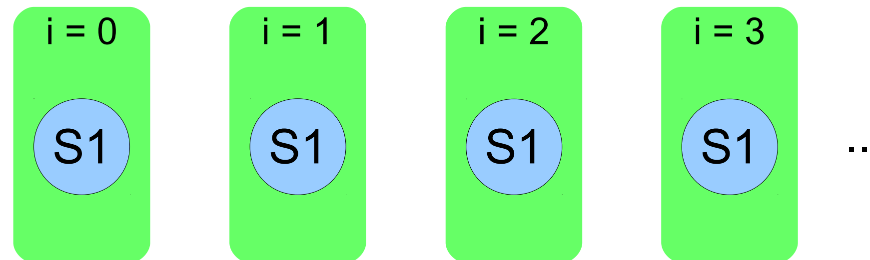


Fundamental Theorem of Dependence

- Any reordering transformation that preserves every dependence in a program preserves the meaning of that program
- Recognizing parallel loops (intuitively)
  - Find data dependences in loop
  - No dependences crossing iteration boundary → parallelization of loop's iterations is safe

Example

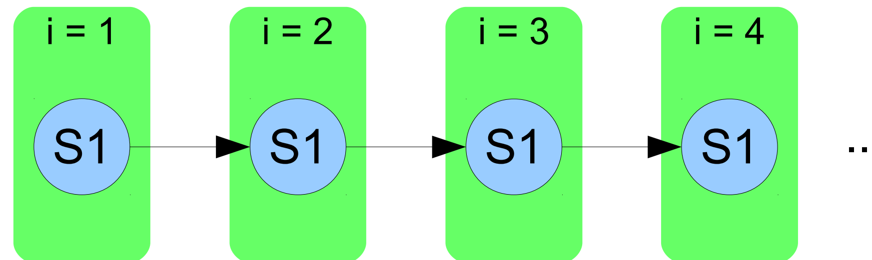
```
for (i=0; i<n; i++) {
S1  a[i] = b[i] + c[i];
}
```



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

Example

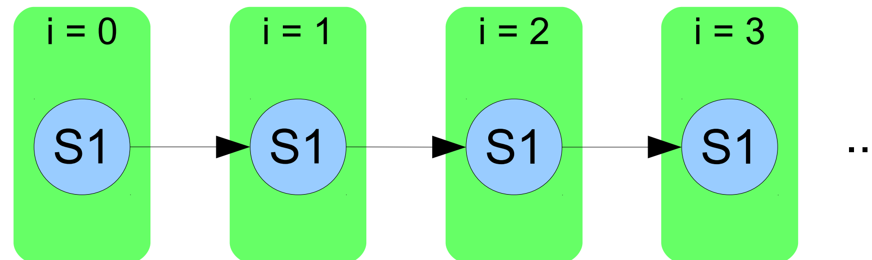
```
for (i=1; i<n; i++) {
S1 a[i] = a[i-1] + b[i]
}
```



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

Example

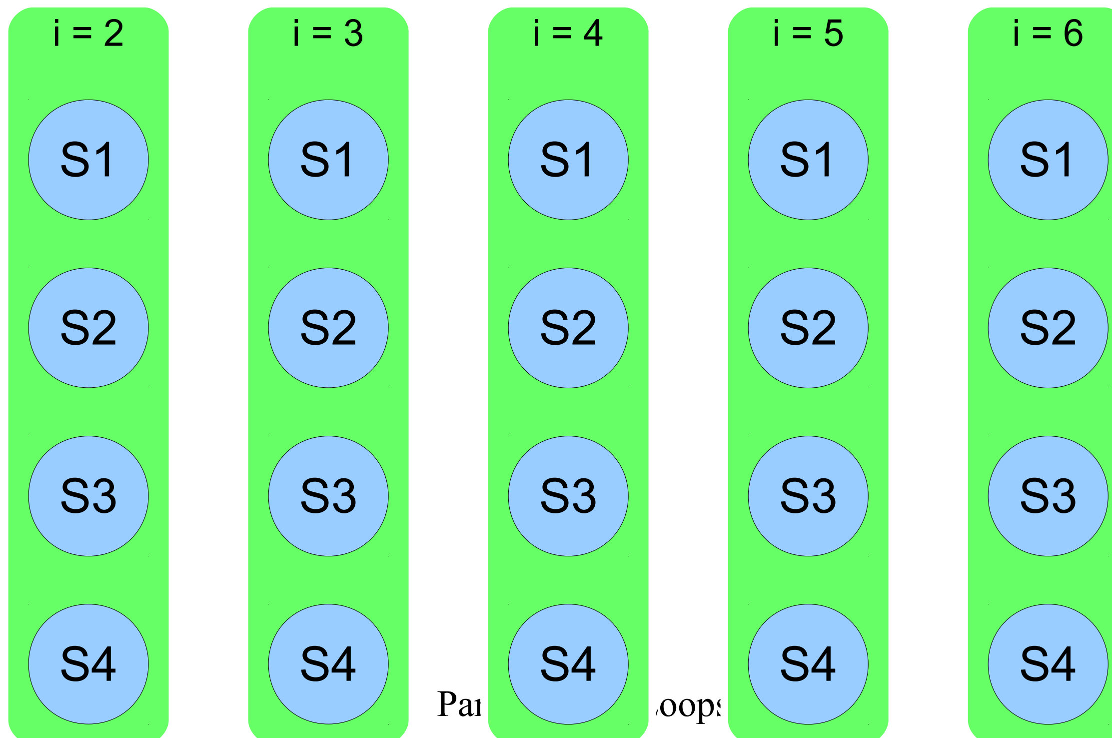
```
s = 0;
for (i=0; i<n; i++) {
S1 s = s + a[i];
}
```



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

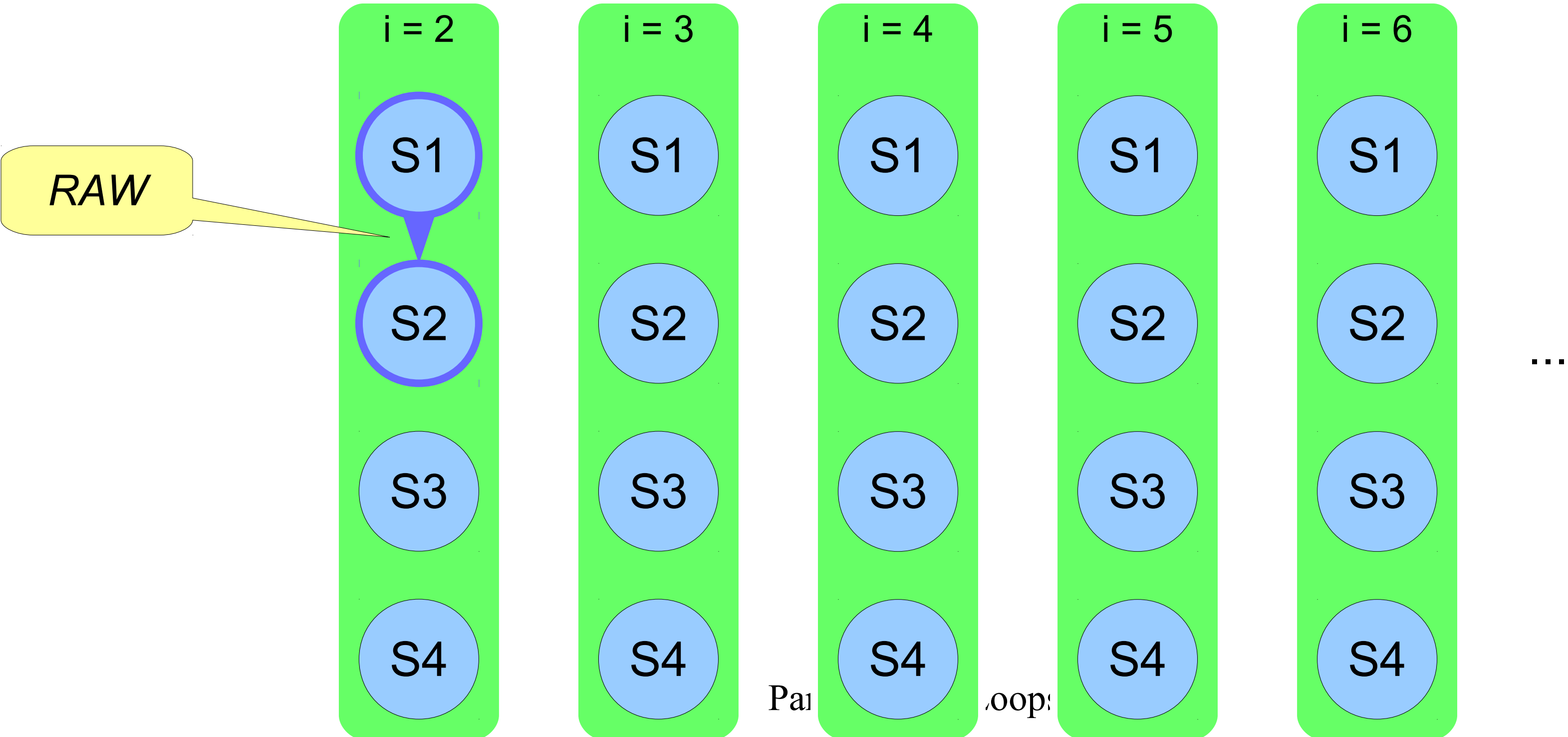
Exercise: draw the dependencies among iterations of the following loop

```
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];
}
```



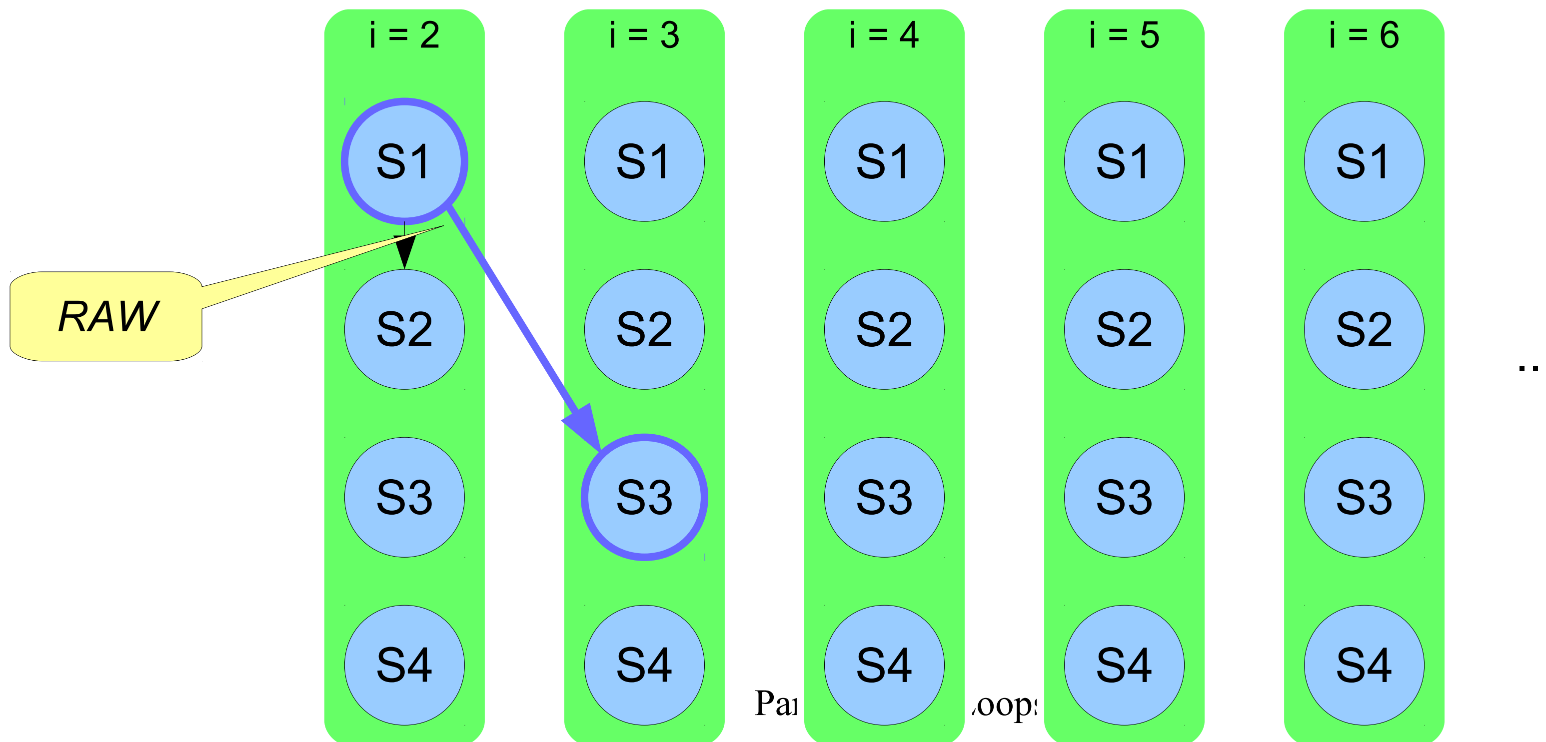
Exercise (cont.)

```
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];
}
```



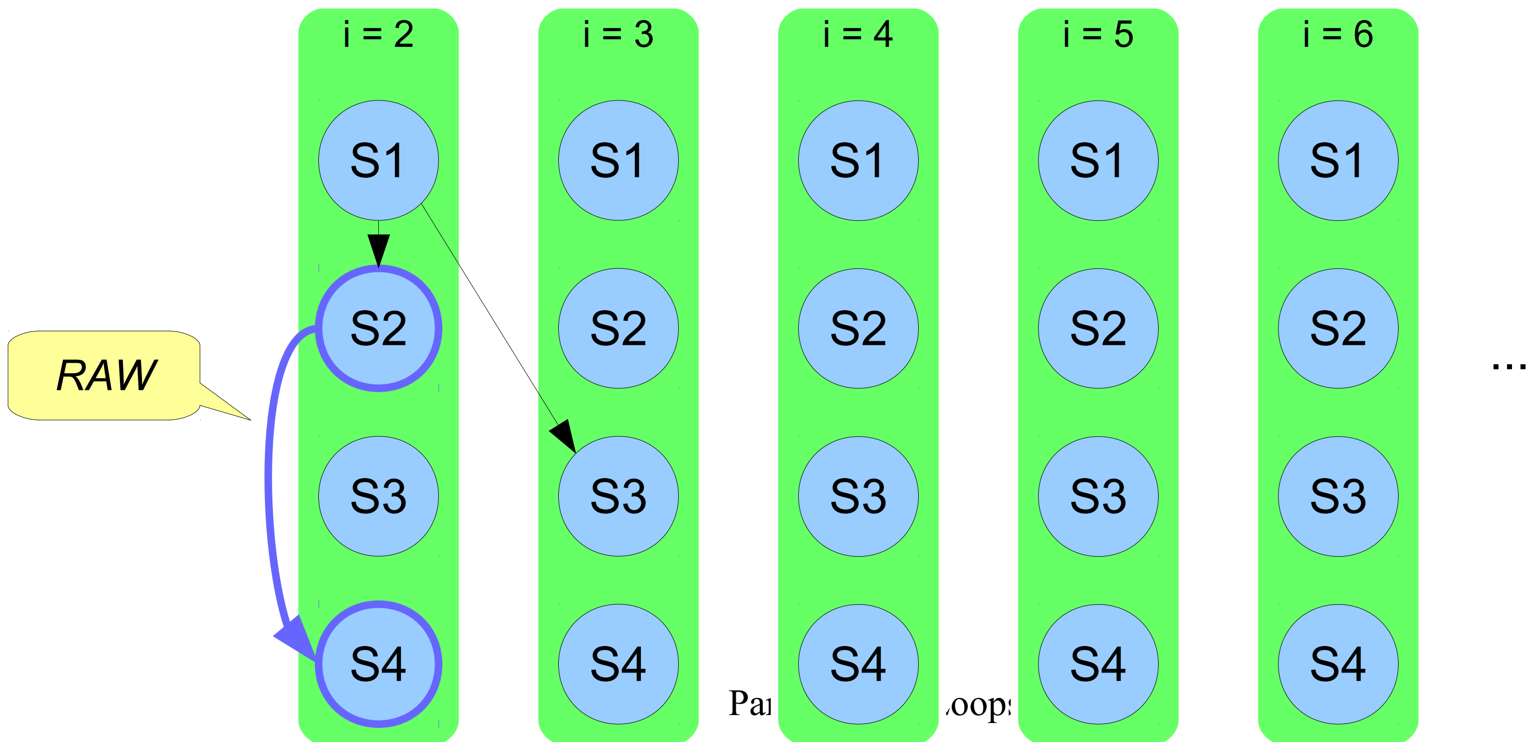
Exercise (cont.)

```
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];
}
```



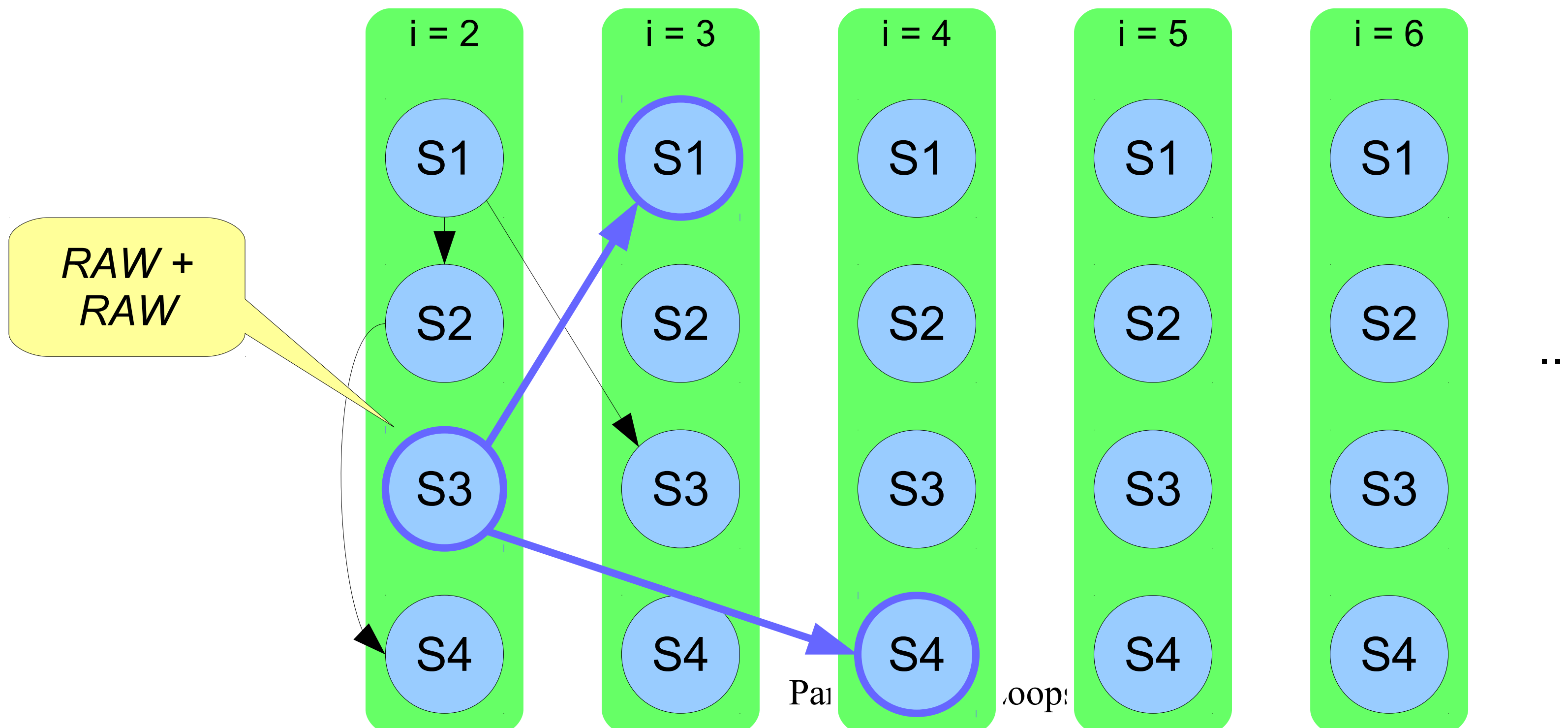
Exercise (cont.)

```
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];
}
```



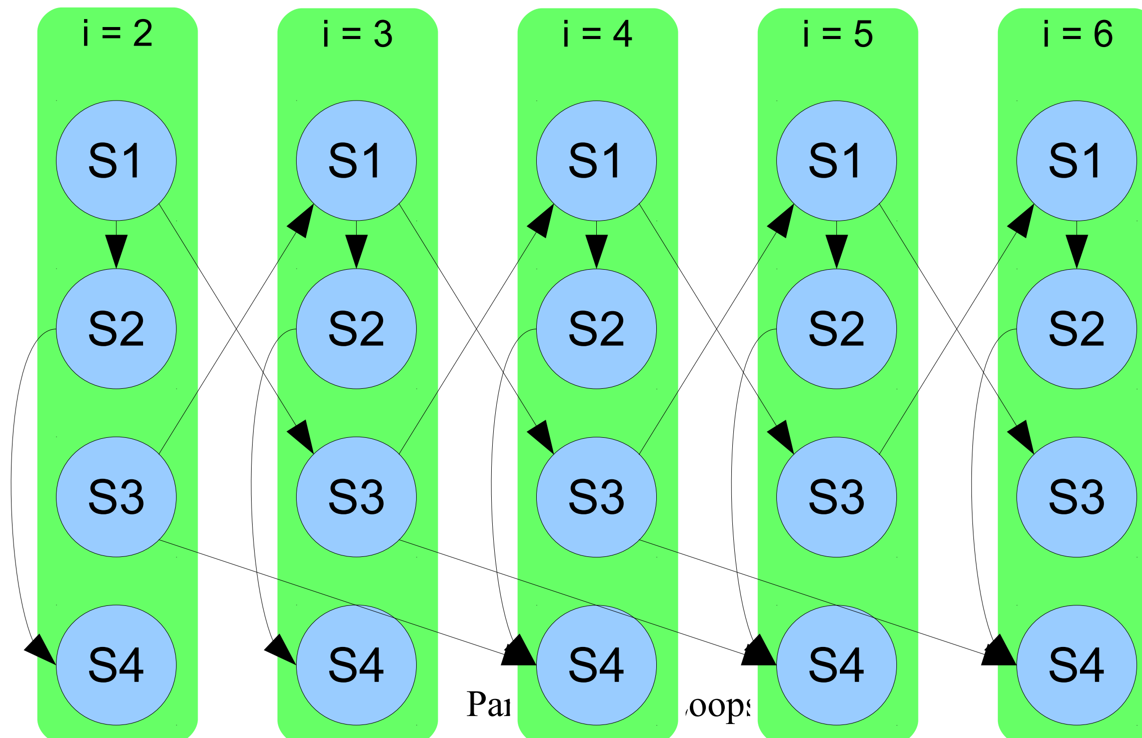
Exercise (cont.)

```
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.)

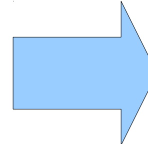
```
for (i=2; i<n; i++) {
S1  a[i] = 4 * c[i-1] - 2;
S2  b[i] = a[i] * 2;
S3  c[i] = a[i-1] + 3;
S4  d[i] = b[i] + c[i-2];
}
```



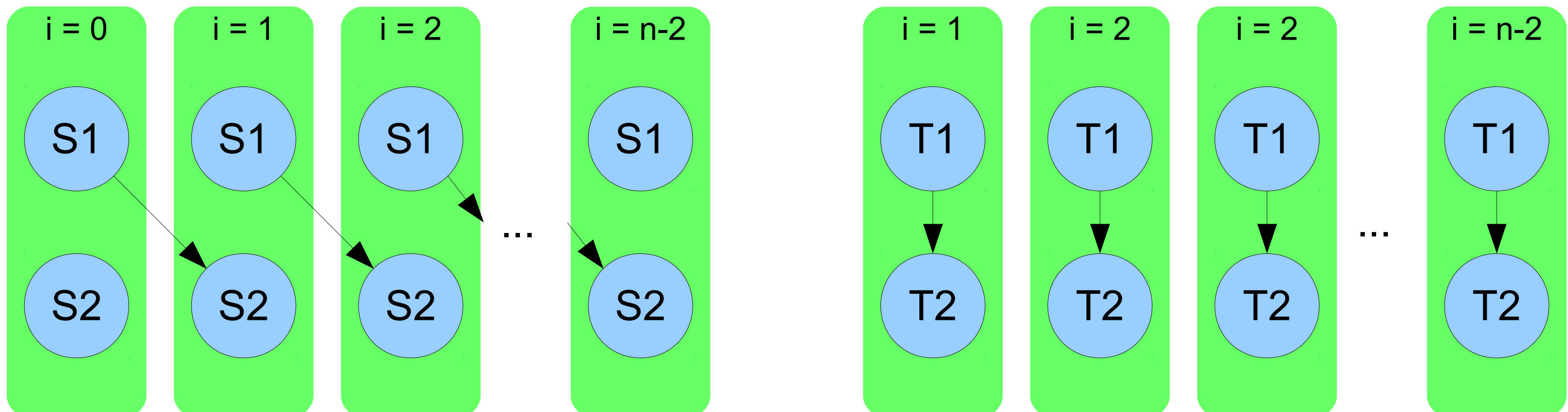
Removing dependencies: Loop aligning

- Dependencies can sometimes be removed by aligning loop iterations

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



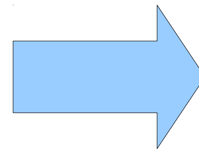
```
a[0] = 0;
d[0] = a[0] + 2;
for (i=1; i<n-1; i++) {
T1  a[i] = b[i-1] * c[i-1];
T2  d[i] = a[i] + 2;
}
a[n-1] = b[n-1] * c[n-1];
```



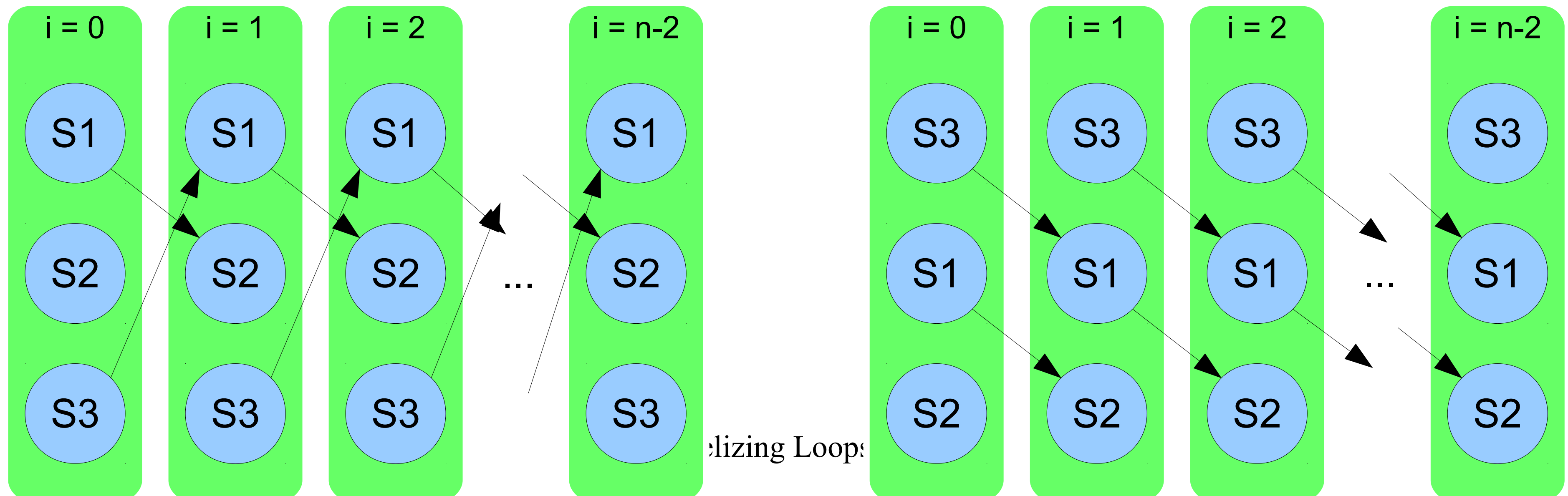
Removing dependencies: Reordering / 1

- Some dependencies can be removed by reordering

```
for (i=0; i<n-1; i++) {
S1  a[i+1] = c[i] + k;
S2  b[i]   = b[i] + a[i];
S3  c[i+1] = d[i] + w;
}
```



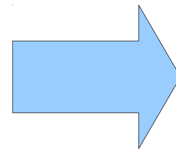
```
for (i=0; i<n-1; i++) {
S3  c[i+1] = d[i] + w;
S1  a[i+1] = c[i] + k;
S2  b[i]   = b[i] + a[i];
}
```



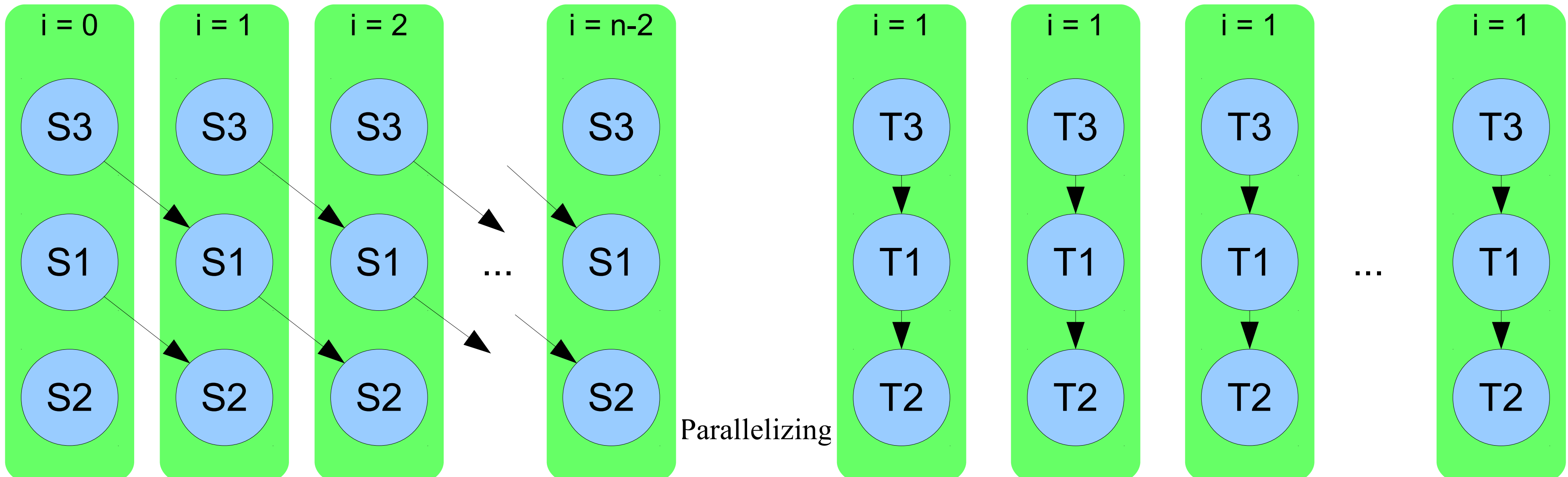
Reordering / 2

- After reordering, we can align loop iterations

```
for (i=0; i<n-1; i++) {
S3  c[i+1] = d[i] + w;
S1  a[i+1] = c[i] + k;
S2  b[i]   = b[i] + a[i];
}
```



```
b[0] = b[0] + a[0];
a[1] = c[0] + k;
b[1] = c[1] + a[1];
for (i=1; i<n-2; i++) {
T3  c[i] = d[i-1] + w;
T1  a[i+1] = c[i] + k;
T2  b[i+1] = b[i+1] + a[i+1];
}
c[n-2] = d[n-3] + w;
a[n-1] = c[n-2] + k;
c[n-1] = d[n-2] + w;
```

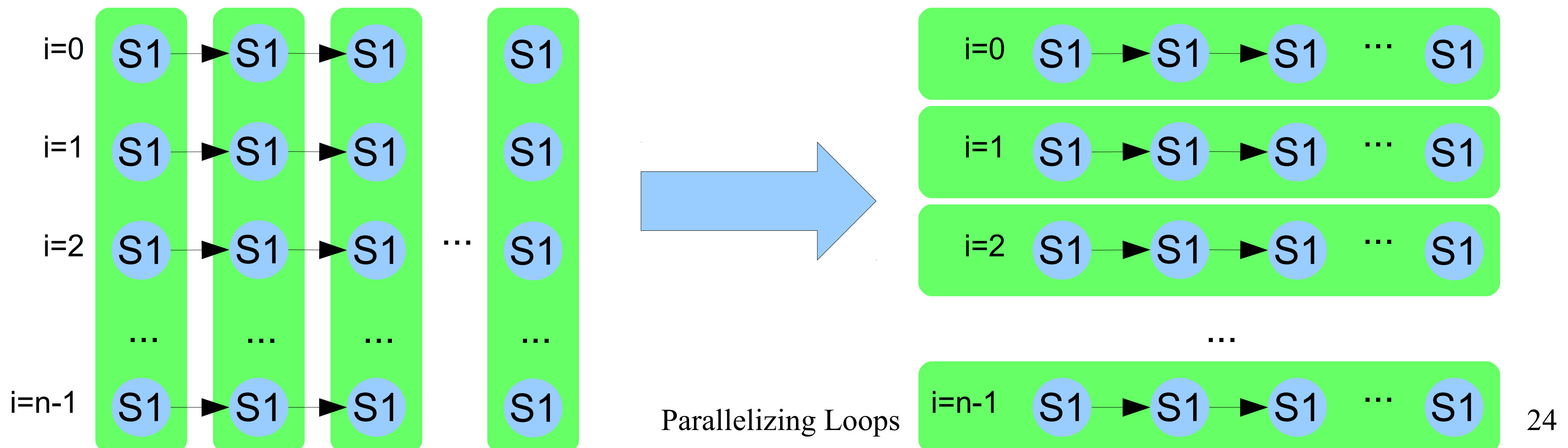


Loop interchange

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

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



Example 1
(Arnold's Cat Map exercise)

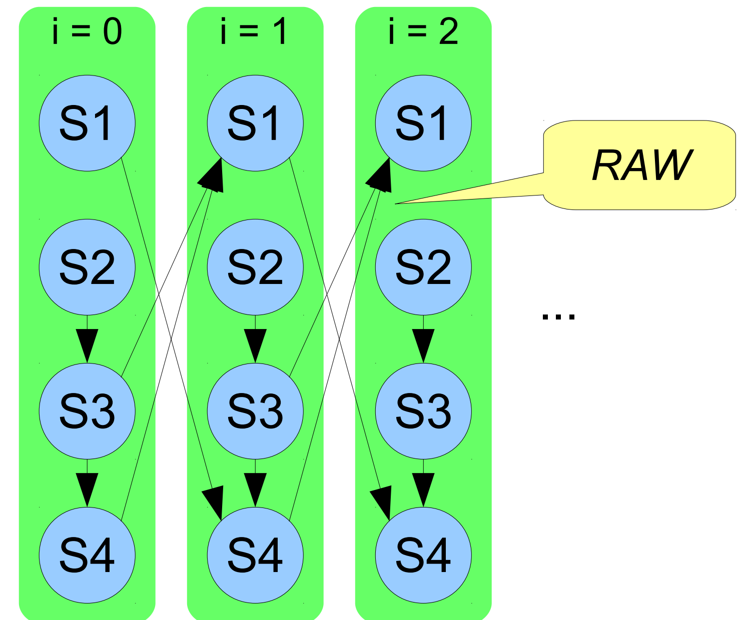
- Which loop(s) can be parallelized?

```
for (i=0; i<k; i++) {
    for (y=0; y<h; y++) {
        for (x=0; x<w; x++) {
            int xnext = (2*x+y) % w;
            int ynext = (x + y) % h;
            next[ynext][xnext] = cur[y][x];
        }
    }
    /* Swap old and new */
    tmp = cur;
    cur = next;
    next = tmp;
}
```

Example 1
(Arnold's Cat Map exercise)

- Which loop(s) can be parallelized?

```
     for (i=0; i<k; i++) {
         for (y=0; y<h; y++) {
             for (x=0; x<w; x++) {
                 int xnext = (2*x+y) % w;
                 int ynext = (x + y) % h;
S1               next[ynext][xnext] = cur[y][x];
             }
         }
         /* Swap old and new */
S2       tmp = cur;
S3       cur = next;
S4       next = tmp;
     }
```



Example 1
(Arnold's Cat Map exercise)

- Which loop(s) can be parallelized?

```
for (i=0; i<k; i++) {
    for (y=0; y<h; y++) {
        for (x=0; x<w; x++) {
S1          int xnext = (2*x+y) % w;
S2          int ynext = (x + y) % h;
S3          next[ynext][xnext] = cur[y][x];
        }
    }
    /* Swap old and new */
    tmp = cur;
    cur = next;
    next = tmp;
}
```

*There are **no dependencies** across loop iterations (there **could** be a WAW dependency on next, but we know that the cat map is bijective so it can't happen). These loops can be **fully parallelized.***

Example 2
(Bellman-Ford Algorithm)

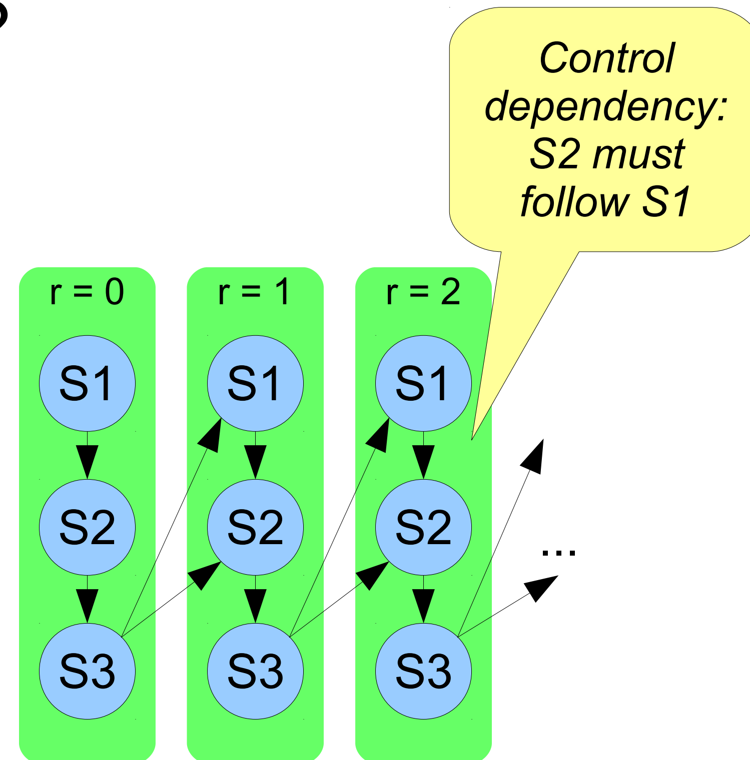
- Which loop(s) can be parallelized?

```
for (r=0; r<n; r++) {
    for (e=0; e<m; e++) {
        const int i = g->edges[e].src;
        const int j = g->edges[e].dst;
        const double wij = g->edges[e].w;
  S1    if (d[i] + wij < d[j]) {
  S2        dnew[j] = d[i] + wij;
        }
    }
    for (i=0; i<n; i++) {
  S3    d[i] = dnew[i];
    }
}
```

Example 2
(Bellman-Ford Algorithm)

- Which loop(s) can be parallelized?

```
for (r=0; r<n; r++) {
    for (e=0; e<m; e++) {
        const int i = g->edges[e].src;
        const int j = g->edges[e].dst;
        const double wij = g->edges[e].w;
S1      if (d[i] + wij < d[j]) {
S2          dnew[j] = d[i] + wij;
        }
    }
    for (i=0; i<n; i++) {
S3      d[i] = dnew[i];
    }
}
```



Example 2
(Bellman-Ford Algorithm)

- Which loop(s) can be parallelized?

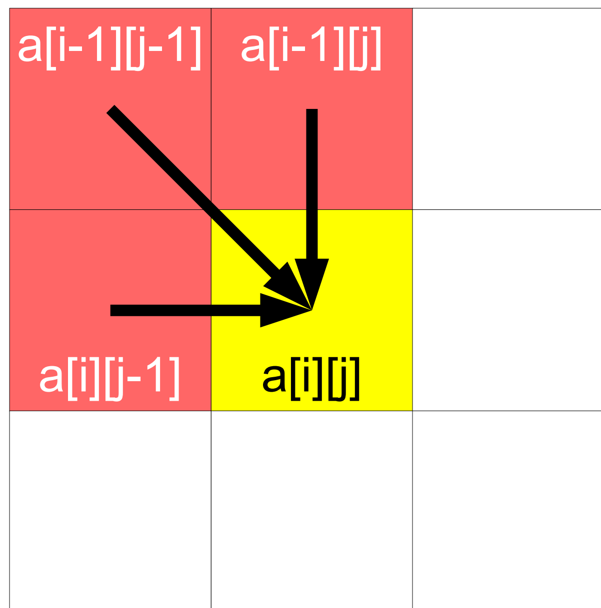
```
for (r=0; r<n; r++) {
    for (e=0; e<m; e++) {
        const int i = g->edges[e].src;
        const int j = g->edges[e].dst;
        const double wij = g->edges[e].w;
S1      if (d[i] + wij < d[j]) {
S2          dnew[j] = d[i] + wij;
        }
    }
    for (i=0; i<n; i++) {
        d[i] = dnew[i];
    }
}
```

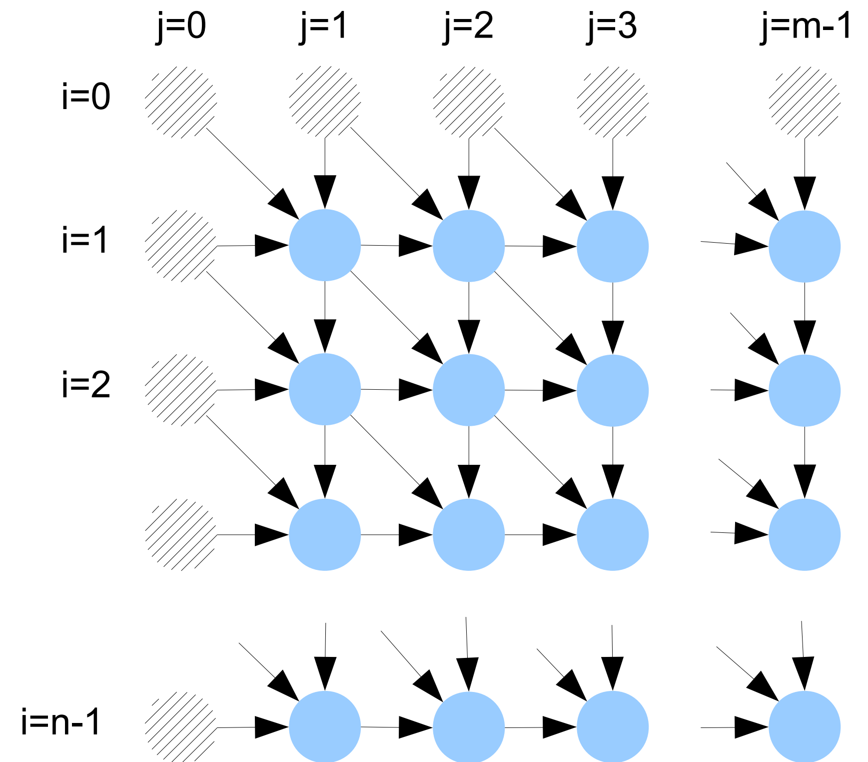
No data dependencies (read d, write dnew)

No data dependencies (read dnew, write d)

Difficult dependencies

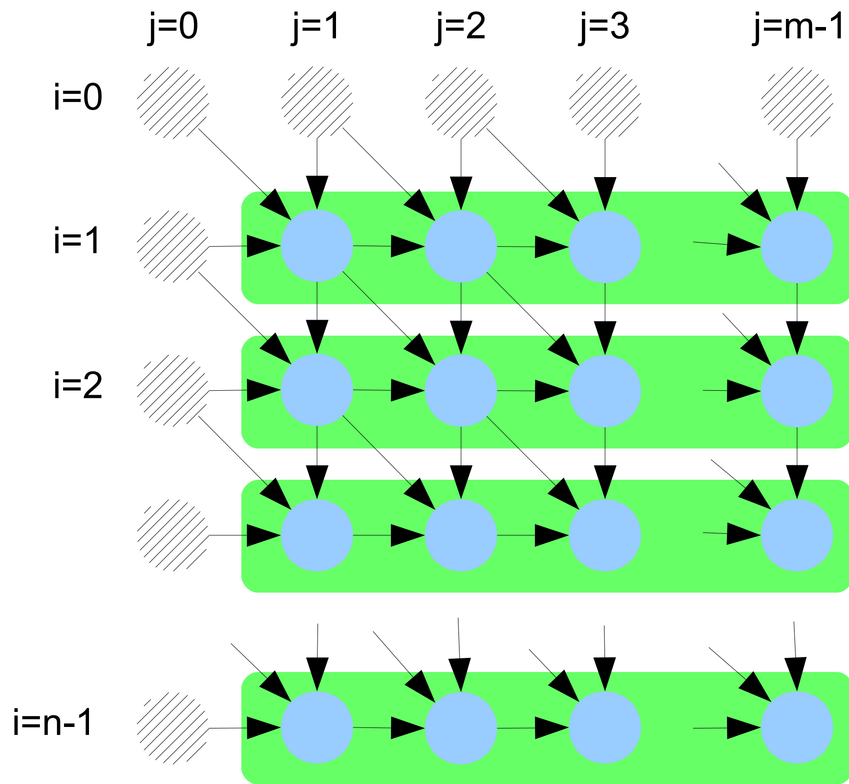
```
for (i=1; i<n; i++) {
  for (j=1; j<m; j++) {
    a[i][j] = a[i-1][j-1] +
              a[i-1][j] +
              a[i][j-1];
  }
}
```

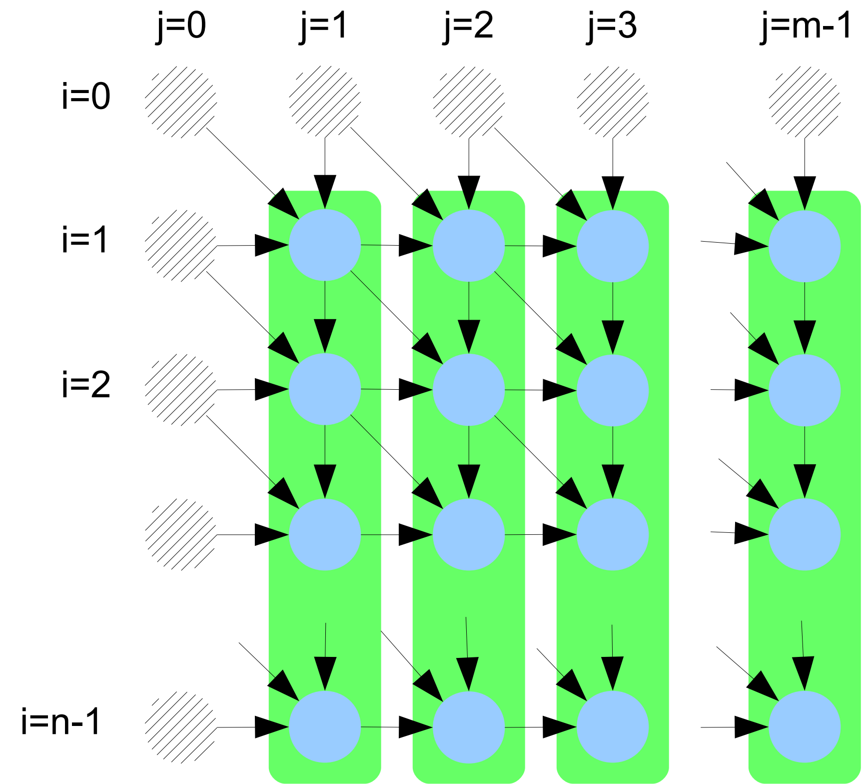




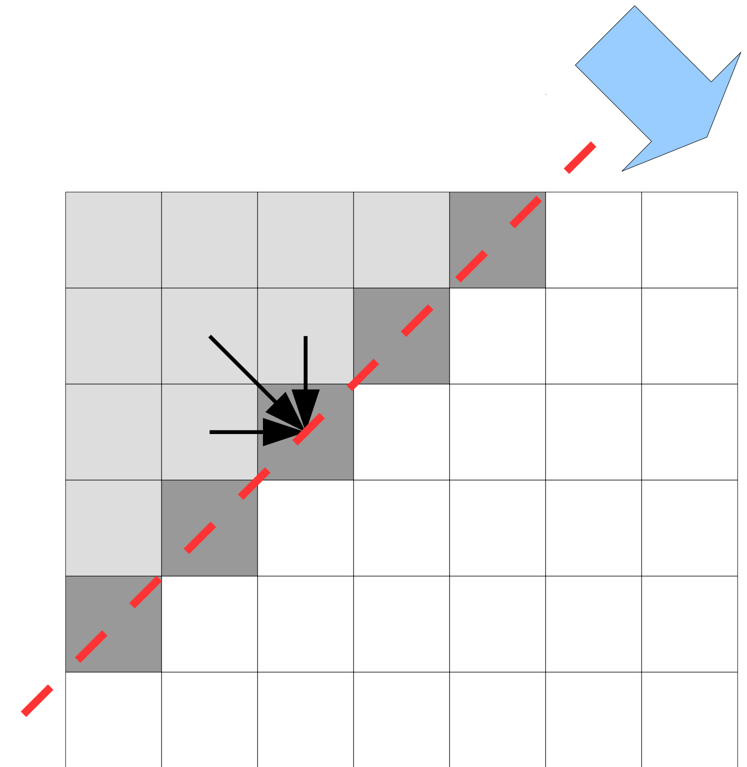
Difficult dependencies

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





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

Conclusions

- A loop can be parallelized if there are no dependencies crossing loop iterations
- Some kinds of dependencies can be eliminated using different algorithms
  - E.g., reductions
- Other kinds of dependencies can be eliminated by sweeping across loop iterations "diagonally"
- Unfortunately, there are situations where dependencies can not be removed, no matter what