Programmazione di Sistemi ~~Embedded e~~ Multicore

Teacher: Daniele De Sensi

Recap

Recap

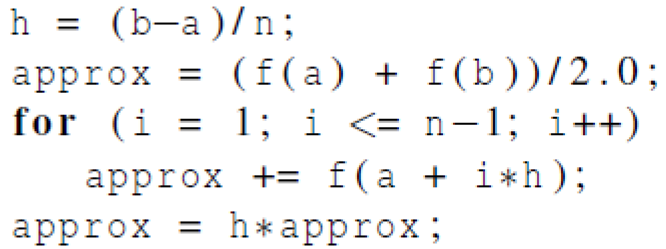
- OpenMP basics
- OpenMP scope

Parallel For

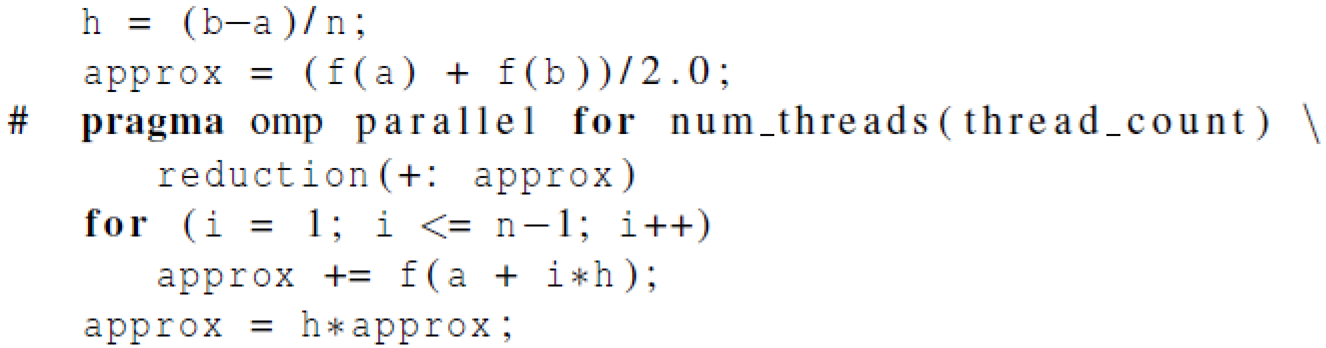
Parallel for

- Forks a team of threads to execute the following structured block.
- However, the structured block following the parallel for directive must be a for loop.
- Furthermore, with the parallel for directive the system parallelizes the for loop by dividing the iterations of the loop among the threads.

Trapezoid Example

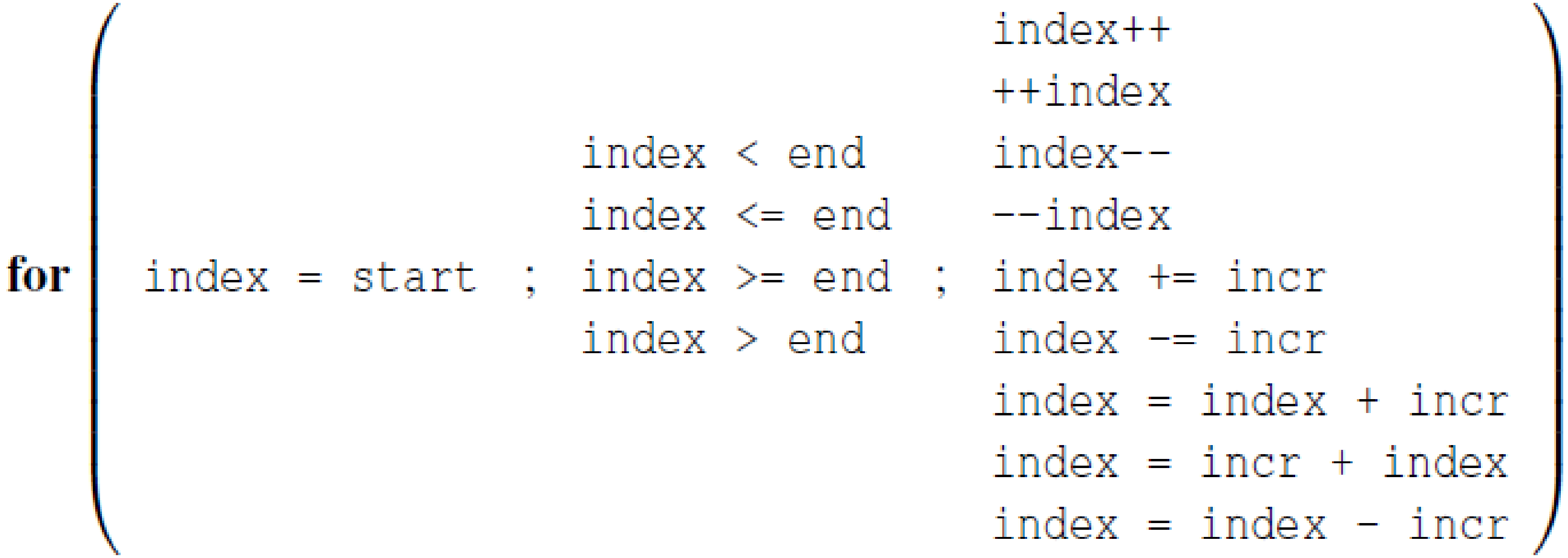


```
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++)
   approx += f(a + i*h);
approx = h*approx;
```



```
    h = (b-a)/n;
    approx = (f(a) + f(b))/2.0;
#   pragma omp parallel for num_threads(thread_count) \
       reduction(+: approx)
    for (i = 1; i <= n-1; i++)
       approx += f(a + i*h);
    approx = h*approx;
```

Legal forms for parallelizable for statements



$$
\textbf{for}\left(\begin{array}{lclcl}
 & & & & \texttt{index++} \\
 & & & & \texttt{++index} \\
 & & \texttt{index < end} & & \texttt{index--} \\
 & & \texttt{index <= end} & & \texttt{--index} \\
\texttt{index = start} & ; & \texttt{index >= end} & ; & \texttt{index += incr} \\
 & & \texttt{index > end} & & \texttt{index -= incr} \\
 & & & & \texttt{index = index + incr} \\
 & & & & \texttt{index = incr + index} \\
 & & & & \texttt{index = index - incr}
\end{array}\right)
$$

Why? It allows the runtime system to determine the number of iterations prior to the execution of the loop

Caveats

- The variable index must have integer or pointer type (e.g., it can't be a float).

- The expressions start, end, and incr must have a compatible type. For example, if index is a pointer, then incr must have integer type.

- The expressions start, end, and incr must not change during execution of the loop.

- During execution of the loop, the variable index can only be modified by the "increment expression" in the for statement.

examples

```
for (i=0; i<n; i++) {
   if (…) break;     //cannot be parallelized
}
```

```
for (i=0; i<n; i++) {
   if (…) return 1; //cannot be parallelized
}
```

```
for (i=0; i<n; i++) {
   if (…) exit();     //can be parallelized
}
```

```
for (i=0; i<n; i++) {
   if (…) i++;     //CANNOT be parallelized
}
```

Questions?

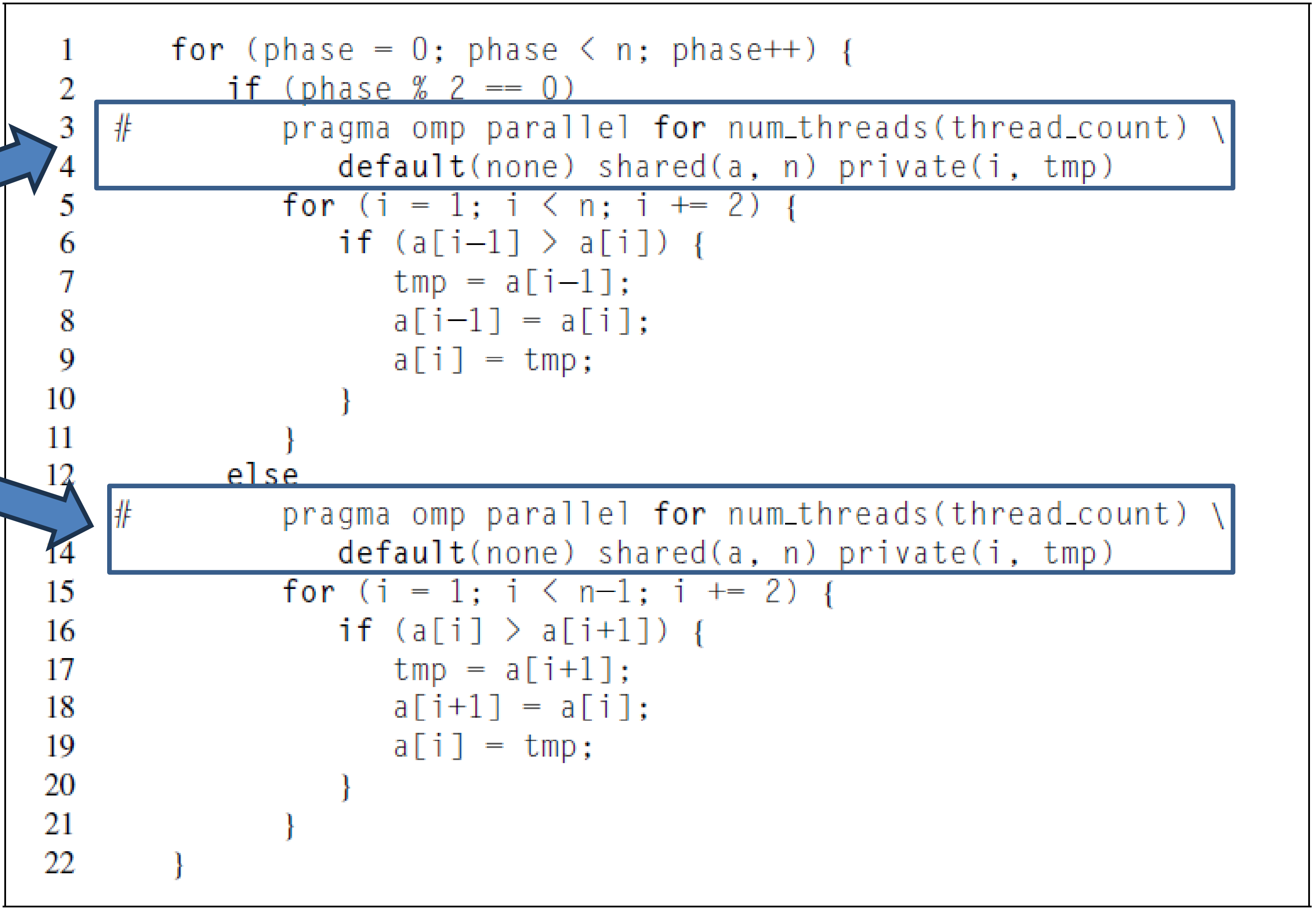
Example: Odd-Even Sort

Odd-Even Sort

This might fork/join new threads everytime it is called (depends on the implementation)

If it does so, we would have some **overhead**

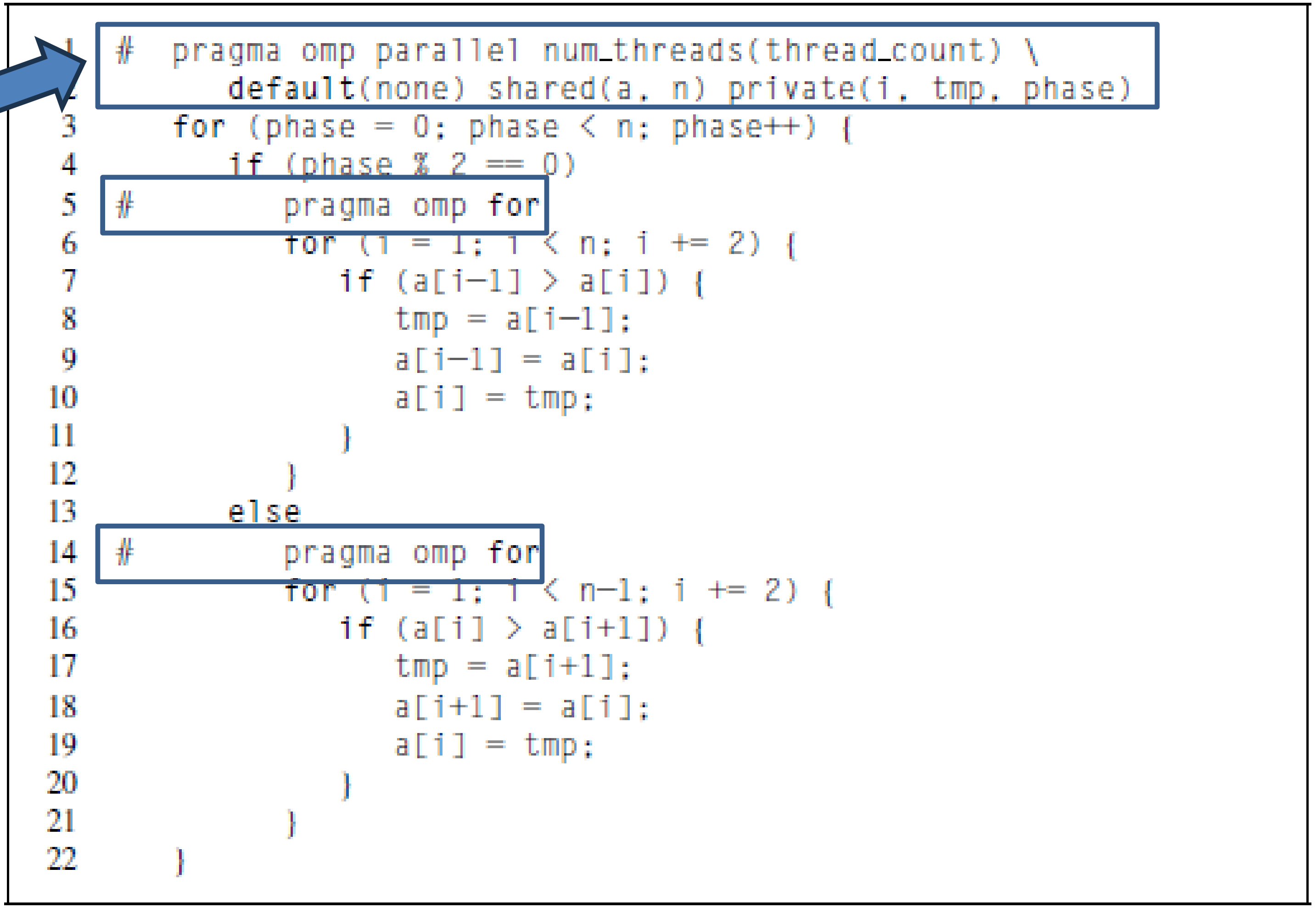
Can we just create the threads at the beginning (before line 1)?



```
1      for (phase = 0; phase < n; phase++) {
2         if (phase % 2 == 0)
3   #           pragma omp parallel for num_threads(thread_count) \
4                  default(none) shared(a, n) private(i, tmp)
5               for (i = 1; i < n; i += 2) {
6                  if (a[i−1] > a[i]) {
7                     tmp = a[i−1];
8                     a[i−1] = a[i];
9                     a[i] = tmp;
10                 }
11              }
12        else
13  #           pragma omp parallel for num_threads(thread_count) \
14                 default(none) shared(a, n) private(i, tmp)
15              for (i = 1; i < n−1; i += 2) {
16                 if (a[i] > a[i+1]) {
17                    tmp = a[i+1];
18                    a[i+1] = a[i];
19                    a[i] = tmp;
20                 }
21              }
22     }
```

Odd-Even Sort

Fork threads only here



```
#  pragma omp parallel num_threads(thread_count) \
      default(none) shared(a, n) private(i, tmp, phase)
   for (phase = 0; phase < n; phase++) {
      if (phase % 2 == 0)
#        pragma omp for
         for (i = 1; i < n; i += 2) {
            if (a[i-1] > a[i]) {
               tmp = a[i-1];
               a[i-1] = a[i];
               a[i] = tmp;
            }
         }
      else
#        pragma omp for
         for (i = 1; i < n-1; i += 2) {
            if (a[i] > a[i+1]) {
               tmp = a[i+1];
               a[i+1] = a[i];
               a[i] = tmp;
            }
         }
   }
```

Odd-Even Sort

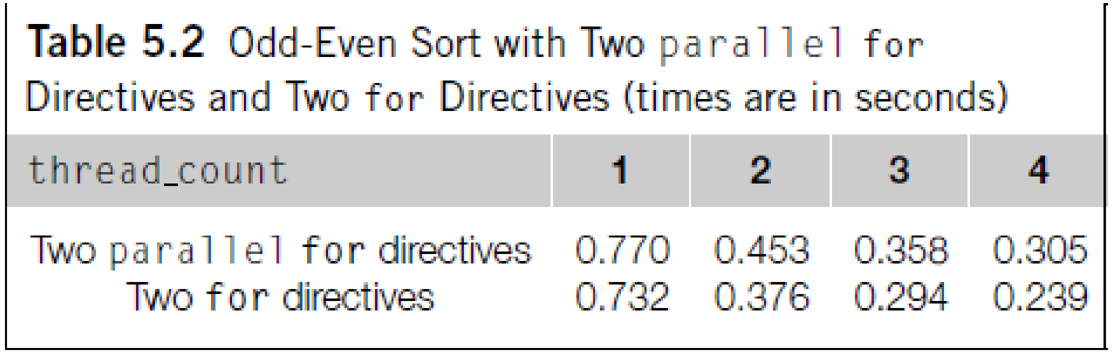


Table 5.2 Odd-Even Sort with Two parallel for Directives and Two for Directives (times are in seconds)

thread_count	1	2	3	4
Two parallel for directives	0.770	0.453	0.358	0.305
Two for directives	0.732	0.376	0.294	0.239

Reusing the same threads provide faster execution times

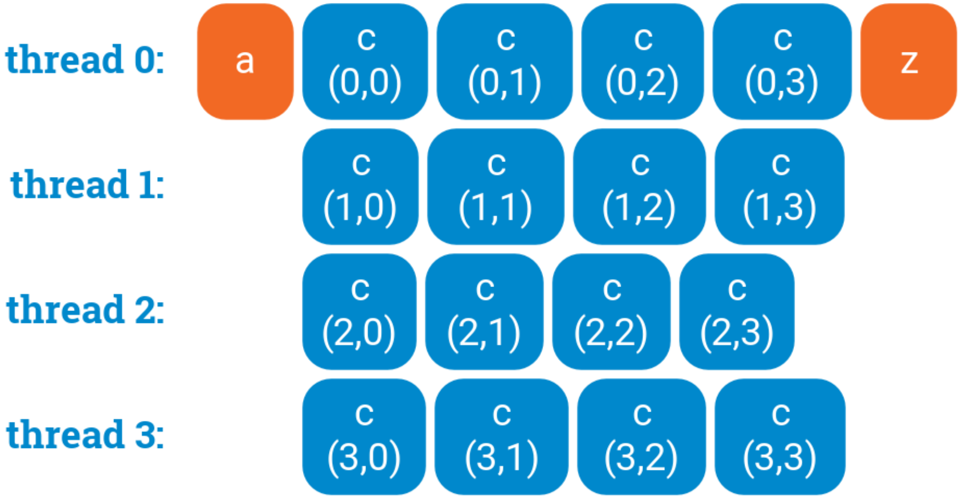
Questions?

Nested Loops

Nested for loops

- If we have nested for loops, it is often enough to simply parallelize the outermost loop

```
a();
#pragma omp parallel for
for (int i = 0; i < 4; ++i) {
    for (int j = 0; j < 4; ++j) {
        c(i, j);
    }
}
z();
```



thread 0: a c(0,0) c(0,1) c(0,2) c(0,3) z

thread 1: c(1,0) c(1,1) c(1,2) c(1,3)

thread 2: c(2,0) c(2,1) c(2,2) c(2,3)

thread 3: c(3,0) c(3,1) c(3,2) c(3,3)

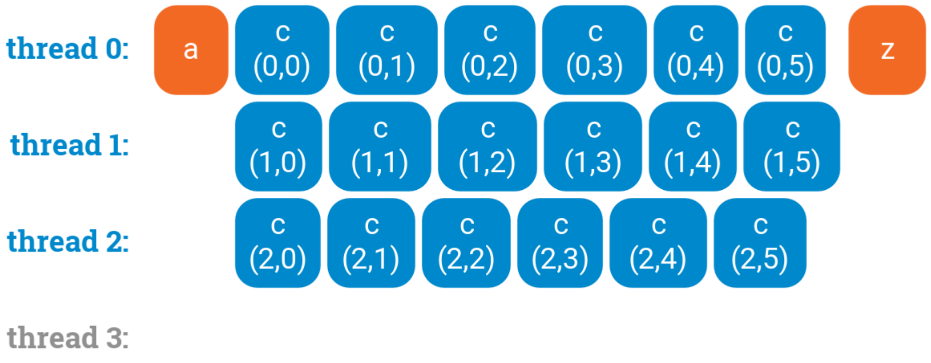
https://ppc.cs.aalto.fi/ch3/nested/

Nested for loops

- Sometimes the outermost loop is so short that not all threads are utilized:

3 iterations, so it won't have sense to start more than 3 threads

```
a();
#pragma omp parallel for
for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 6; ++j) {
        c(i, j);
    }
}
z();
```

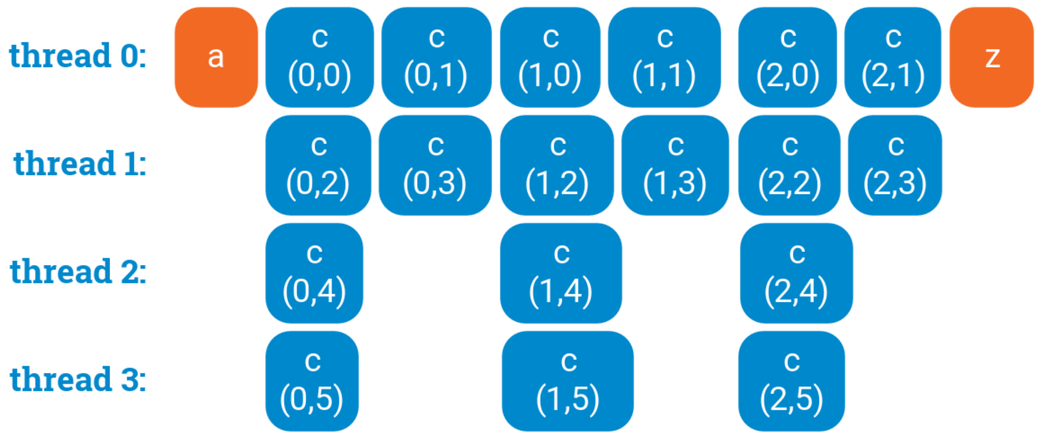


https://ppc.cs.aalto.fi/ch3/nested/

Nested for loops

- We could try to parallelize the inner loop, but there is no guarantee that the thread utilization is better

```
a();
for (int i = 0; i < 3; ++i) {
#pragma omp parallel for
    for (int j = 0; j < 6; ++j) {
        c(i, j);
    }
}
z();
```



https://ppc.cs.aalto.fi/ch3/nested/

Nested for loops

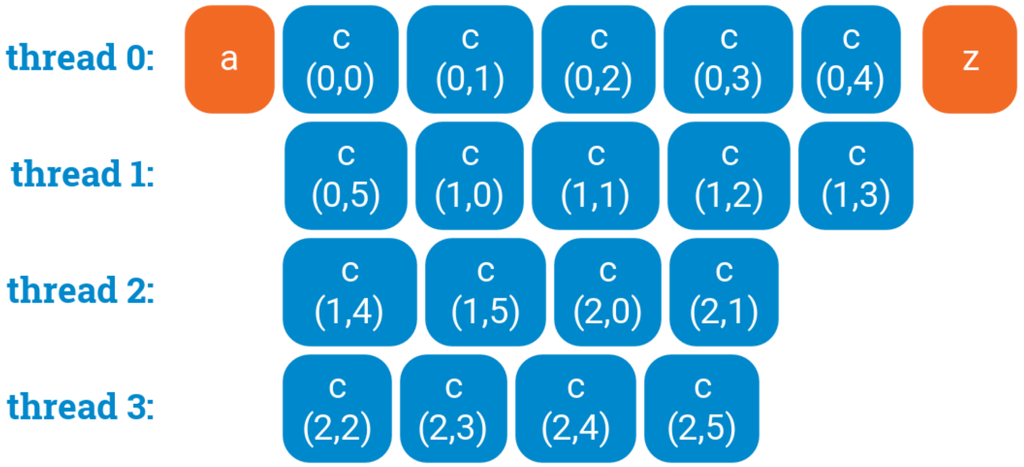
- The correct solution is to collapse it into one loop that does 18 iterations. We can do it manually:

https://ppc.cs.aalto.fi/ch3/nested/

Nested for loops

- The correct solution is to collapse it into one loop that does 18 iterations. We can do it manually:

```
a();
#pragma omp parallel for
for (int ij = 0; ij < 3*6; ++ij) {
        c(ij / 6, ij % 6);
    }
}
z();
```



thread 0: a c (0,0) c (0,1) c (0,2) c (0,3) c (0,4) z

thread 1: c (0,5) c (1,0) c (1,1) c (1,2) c (1,3)

thread 2: c (1,4) c (1,5) c (2,0) c (2,1)

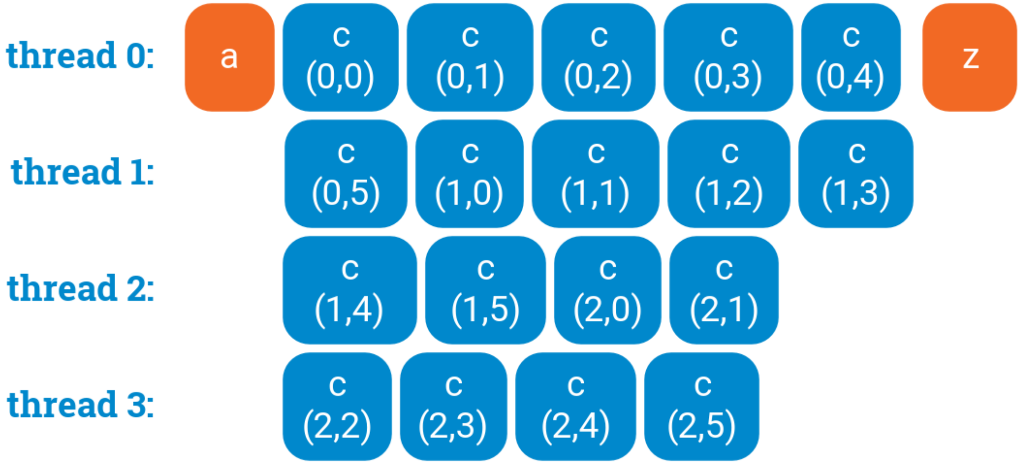
thread 3: c (2,2) c (2,3) c (2,4) c (2,5)

https://ppc.cs.aalto.fi/ch3/nested/

Nested for loops

- we can ask OpenMP to do it for us:

```
a();
#pragma omp parallel for
collapse(2)
for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 6; ++j)
        c(i, j);
}
z();
```



thread 0: a c (0,0) c (0,1) c (0,2) c (0,3) c (0,4) z

thread 1: c (0,5) c (1,0) c (1,1) c (1,2) c (1,3)

thread 2: c (1,4) c (1,5) c (2,0) c (2,1)

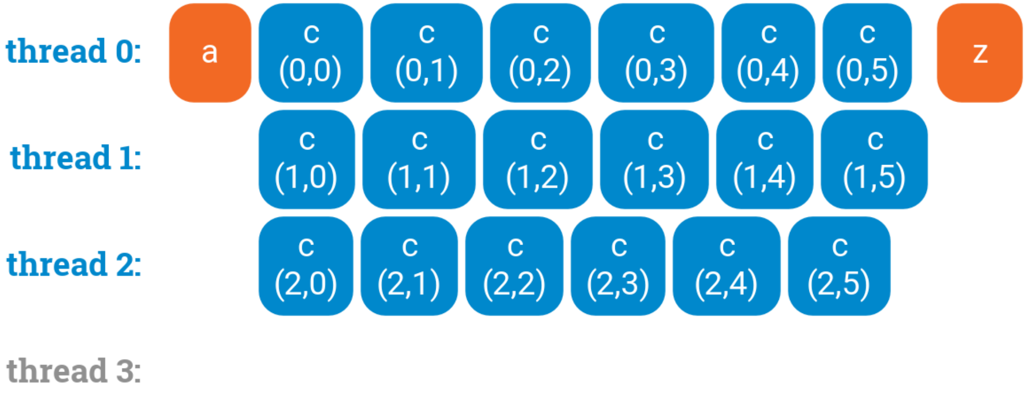
thread 3: c (2,2) c (2,3) c (2,4) c (2,5)

https://ppc.cs.aalto.fi/ch3/nested/

Nested for loops

- Wrong way: "Nested parallelism" is disabled in OpenMP by default (i.e., inner parallel for pragmas will be ignored)

```
a();
#pragma omp parallel for
for (int i = 0; i < 3; ++i) {
#pragma omp parallel for
    for (int j = 0; j < 6; ++j)
        c(i, j);
}
z();
```



https://ppc.cs.aalto.fi/ch3/nested/

Nested for loops

- Wrong way: If "Nested parallelism" is enabled it will create 12 threads on a server with 4 cores (3*4)!

```
a();
#pragma omp parallel for
for (int i = 0; i < 3; ++i) {
#pragma omp parallel for
    for (int j = 0; j < 6; ++j)
      c(i, j);
}
z();
```

https://ppc.cs.aalto.fi/ch3/nested/

Data Dependencies

Chapter 4.4.1 from “Multicore and GPU Programming An Integrated Approach”

Data dependencies

```
fibo[ 0 ] = fibo[ 1 ] = 1;
for (i = 2; i < n; i++)
    fibo[ i ] = fibo[ i - 1 ] + fibo[ i - 2 ];
```

```
 fibo[ 0 ] = fibo[ 1 ] = 1;
#  pragma omp parallel for num_threads(2)
   for (i = 2; i < n; i++)
       fibo[ i ] = fibo[ i - 1 ] + fibo[ i - 2 ];
```

```
1 1 2 3 5 8 13 21 34 55
```

this is correct

but sometimes we get this

```
1 1 2 3 5 8 0 0 0 0
```

What happened?

1. OpenMP compilers don’t check for dependences among iterations in a loop that’s being parallelized with a parallel for directive.

2. A loop in which the results of one or more iterations depend on other iterations cannot, in general, be correctly parallelized by OpenMP. We say that we have a **loop-carried dependence**

Data dependencies

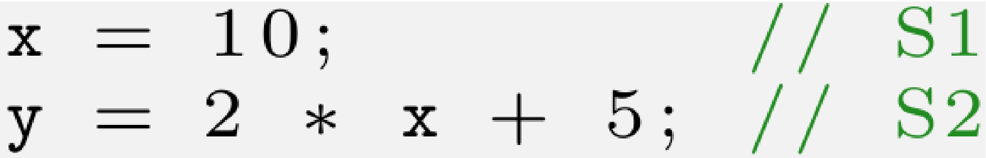
- Assuming we have a loop of the form:

```
for(i=...
{
  S1 : operate on a memory location x
  ...
  S2 : operate on a memory location x
}
```

- There are four different ways that S1 and S2 are connected, based on whether they are reading of writing to x.
- A problem exists if the dependence crosses loop iterations : loop-carried dependence.

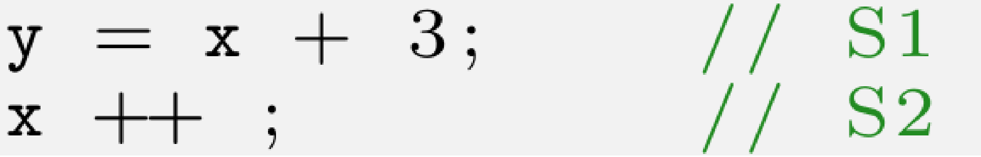
Dependence Types

- Flow dependence : RAW



```
x = 10;          // S1
y = 2 * x + 5;   // S2
```

- Anti-flow dependence : WAR



```
y = x + 3;       // S1
x ++ ;           // S2
```

Dependence Types (cont.)

- Output dependence : WAW

```
x = 10;        // S1
x = x + c ;    // S2
```

- Input dependence : RAR (it's not an actual dependence)

```
y = x + c;        // S1
z = 2 * x + 1;    // S2
```

Questions?

Flow Dependence Removal (RAW)

Data Dependency Resolution

6 techniques:

1. reduction/induction variable fix
2. Loop skewing
3. Partial parallelization
4. Refactoring
5. Fissioning
6. Algorithm change

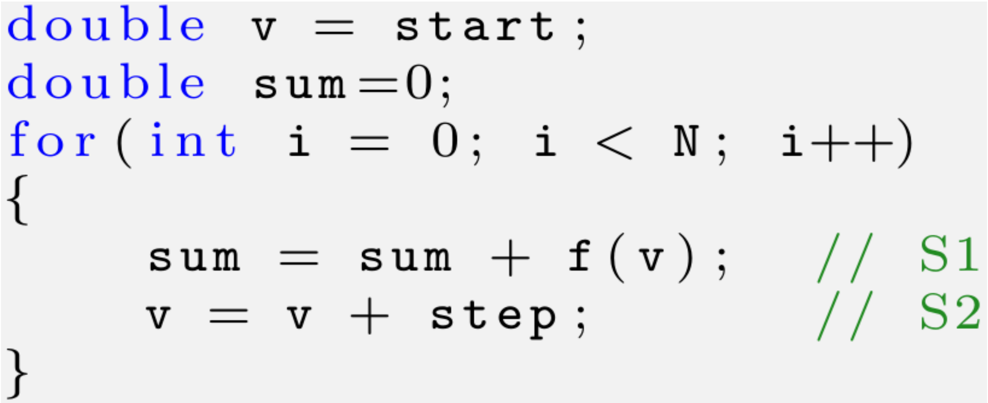
Data Dependency Resolution

6 techniques:

1. reduction/induction variable fix
2. Loop skewing
3. Partial parallelization
4. Refactoring
5. Fissioning
6. Algorithm change

Flow Dependence : Reduction, Induction Variables

- Example:



```
double v = start;
double sum=0;
for(int i = 0; i < N; i++)
{
     sum = sum + f(v);    // S1
     v = v + step;        // S2
}
```

- RAW (S1) caused by reduction variable sum.
- RAW (S2) caused by induction variable v (induction variable is a variable that gets increased/decreased by a constant amount at each iteration).
- RAW (S2->S1) caused by induction variable v.
- Induction variable: affine function of the loop variable.

N.B.: RAW are between the *i* and the *i+1* iterations. E.g., sum is **read** in the *(i+1)-th* iteration **after** being **written** in *i-th* iteration

Remove RAW (S2) and RAW (S2->S1)

```
double v = start;
double sum = 0;
for(int i = 0; i < N ; i++)
{
        sum = sum + f(v); // S1
        v = v + step;      // S2
}
```

```
double v;
double sum = 0;
for(int i = 0; i < N ; i++)
{
        v = start + i*step;
        sum = sum + f(v);
}
```

```
i = 0 -> v = start
i = 1 -> v = start + step
i = 2 -> v = (start + step) + step
…
```

Remove RAW (S1)

```
double v;
double sum =0;
#pragma omp parallel for reduction(+ : sum) private(v)
for(int i = 0; i < N ; i++)
{
        v = start + i*step;
        sum = sum + f(v);
}
```

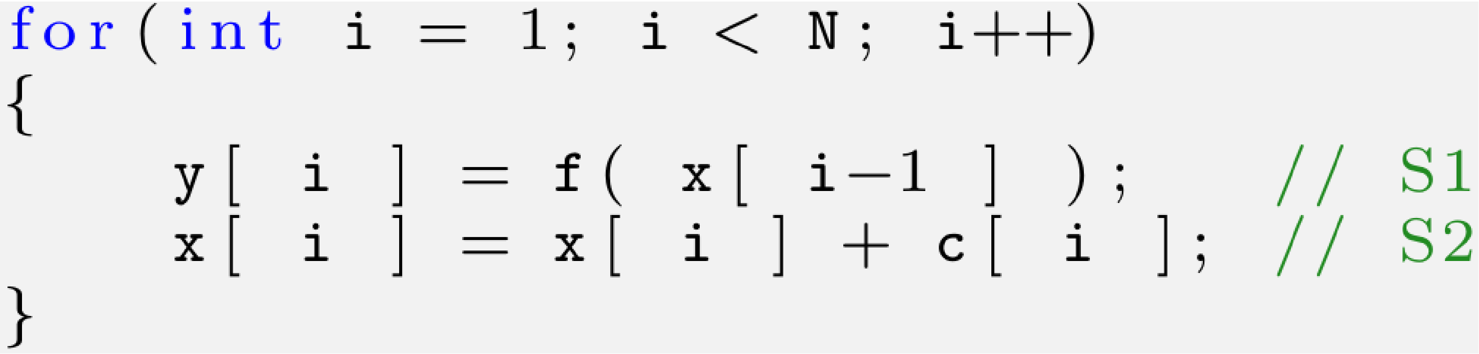
Data Dependency Resolution

6 techniques:

1. reduction/induction variable fix
2. Loop skewing
3. Partial parallelization
4. Refactoring
5. Fissioning
6. Algorithm change

Flow Dependence: Loop Skewing

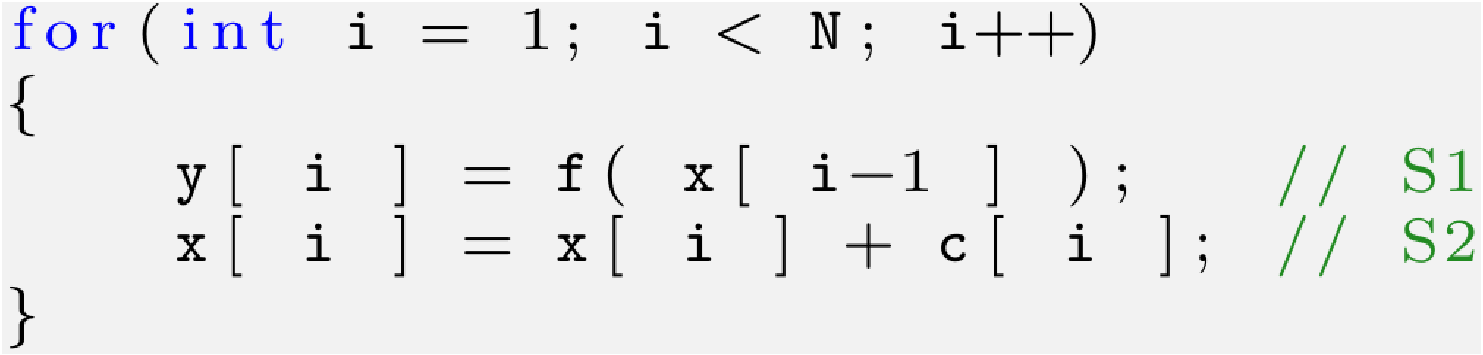
- Another technique involves the rearrangement of the loop body statements. Example with :



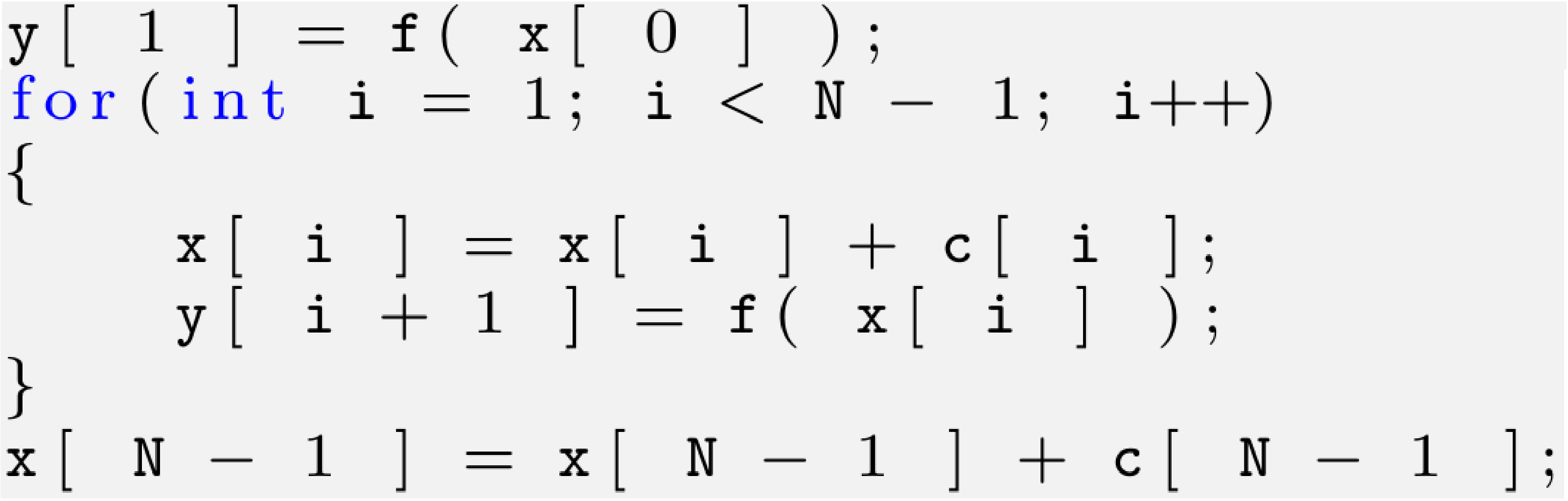
```
for(int i = 1; i < N; i++)
{
      y[ i ] = f( x[ i-1 ] );    // S1
      x[ i ] = x[ i ] + c[ i ];  // S2
}
```

- RAW (S2->S1) on x
- Solution: make sure the statements that consume the calculated values that cause the dependence, use values generated during the same iteration.

Flow Dependence : Loop Skewing (2)



```
for(int i = 1; i < N; i++)
{
    y[ i ] = f( x[ i-1 ] );    // S1
    x[ i ] = x[ i ] + c[ i ]; // S2
}
```



```
y[ 1 ] = f( x[ 0 ] );
for(int i = 1; i < N - 1; i++)
{
    x[ i ] = x[ i ] + c[ i ];
    y[ i + 1 ] = f( x[ i ] );
}
x[ N - 1 ] = x[ N - 1 ] + c[ N - 1 ];
```

Flow Dependence: Loop Skewing (3)

- How to do loop skewing?
  - Hint: unroll the loop and see the repetition pattern

```
for (int i = 1; i< N; i++)
{
        y[i]= f(x[i-1]);
        x[i]= x[i]+c[i];
}
```

```
y[1]= f(x[0]);
x[1]= x[1]+c[1];
y[2]= f(x[1]);
x[2]= x[2]+c[2];
...
y[N-2]= f(x[N-3]);
x[N-2]= x[N-2]+ c[N-2];
y[N-1]= f(x[N-2]);
x[N-1]= x[N-1]+ c[N-1];
```

Flow Dependence: Loop Skewing (4)

- How to do loop skewing?
  - Hint: unroll the loop and see the repetition pattern

```
for (int i = 1; i< N; i++)
{
        y[i]= f(x[i-1]);
        x[i]= x[i]+c[i];
}
```

```
y[1]= f(x[0]);
x[1]= x[1]+c[1];
y[2]= f(x[1]);
x[2]= x[2]+c[2];
...
y[N-2]= f(x[N-3]);
x[N-2]= x[N-2]+ c[N-2];
y[N-1]= f(x[N-2]);
x[N-1]= x[N-1]+ c[N-1];
```

Flow Dependence: Loop Skewing (5)

- How to do loop skewing?
  - Hint: unroll the loop and see the repetition pattern

```
y[1]= f(x[0]);
for (int i = 1; i< N-1; i++)
{
        x[i]= x[i]+c[i];
        y[i+1]= f(x[i]);
}
x[N-1]= x[N-1]+ c[N-1];
```

```
y[1]= f(x[0]);
x[1]= x[1]+c[1];
y[2]= f(x[1]);
x[2]= x[2]+c[2];
...
y[N-2]= f(x[N-3]);
x[N-2]= x[N-2]+ c[N-2];
y[N-1]= f(x[N-2]);
x[N-1]= x[N-1]+ c[N-1];
```

Data Dependency Resolution

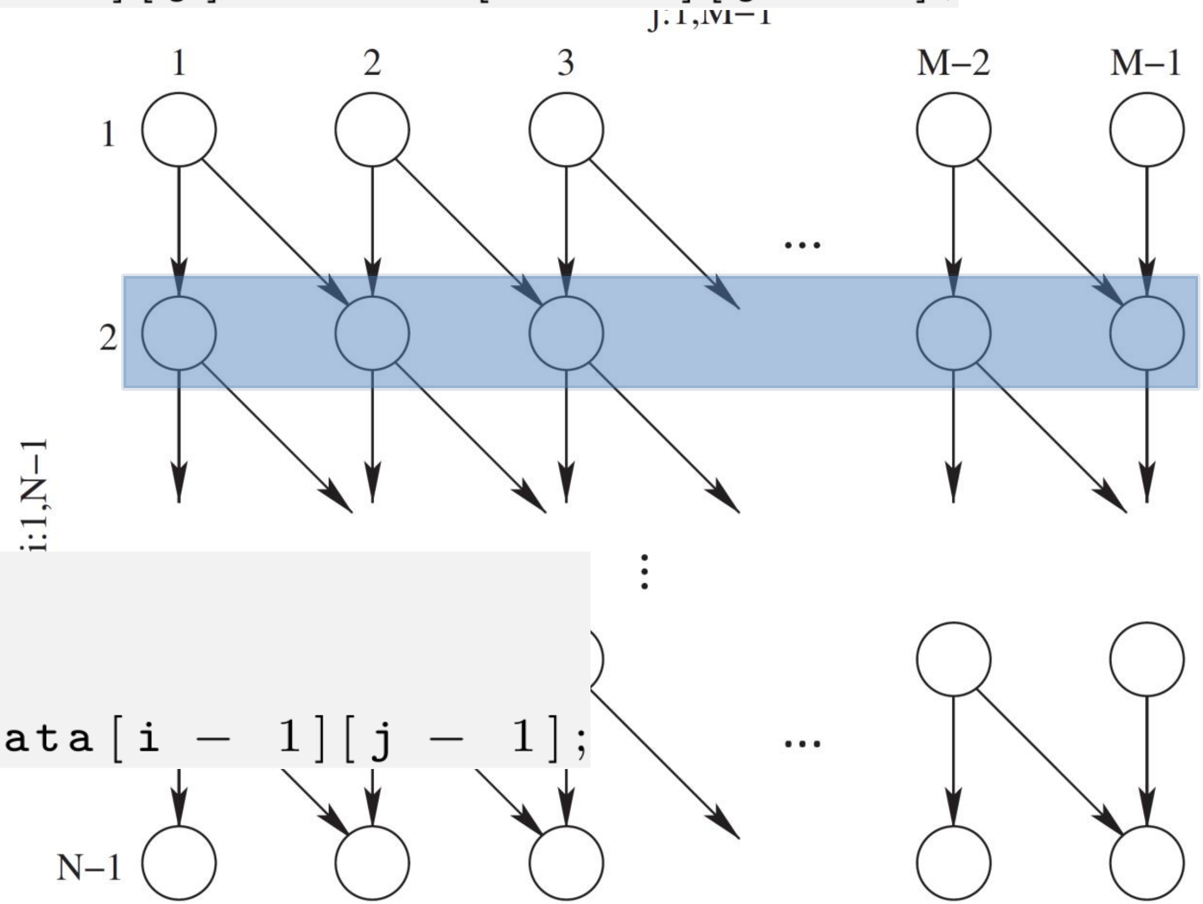
6 techniques:

1. reduction/induction variable fix
2. Loop skewing
3. Partial parallelization
4. Refactoring
5. Fissioning
6. Algorithm change

Iteration Space Dependency Graph

- **ISDG** is made up of nodes that represent an single execution of the loop body, and edges that represent dependencies.
- Example:

```
for (int i = 1; i < N; i++)
  for (int j = 1; j < M; j++)
    data[i][j] = data[i − 1][j] + data[i − 1][j − 1];
```



No edges/dependencies between nodes on the same row.
I.e., we can parallelize the j-loop

```
for (int i = 1; i < N; i++)
#pragma omp parallel for
    for (int j = 1; j < M; j++)
      data[i][j] = data[i − 1][j] + data[i − 1][j − 1];
```

Data Dependency Resolution

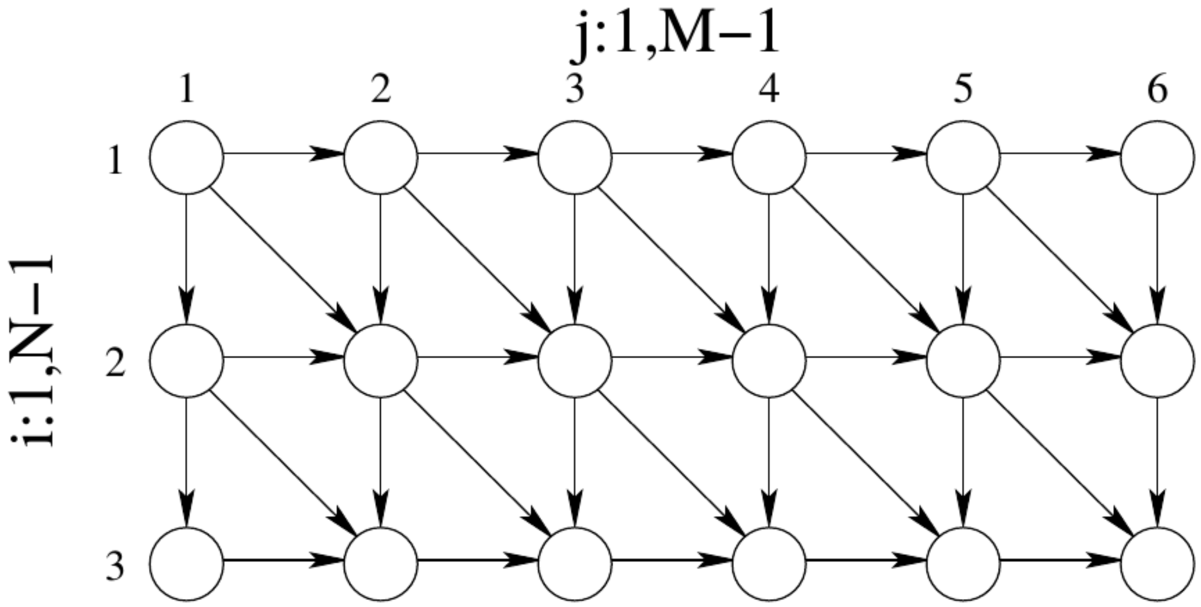
6 techniques:

1. reduction/induction variable fix
2. Loop skewing
3. Partial parallelization
4. Refactoring
5. Fissioning
6. Algorithm change

Flow Dependencies: Refactoring

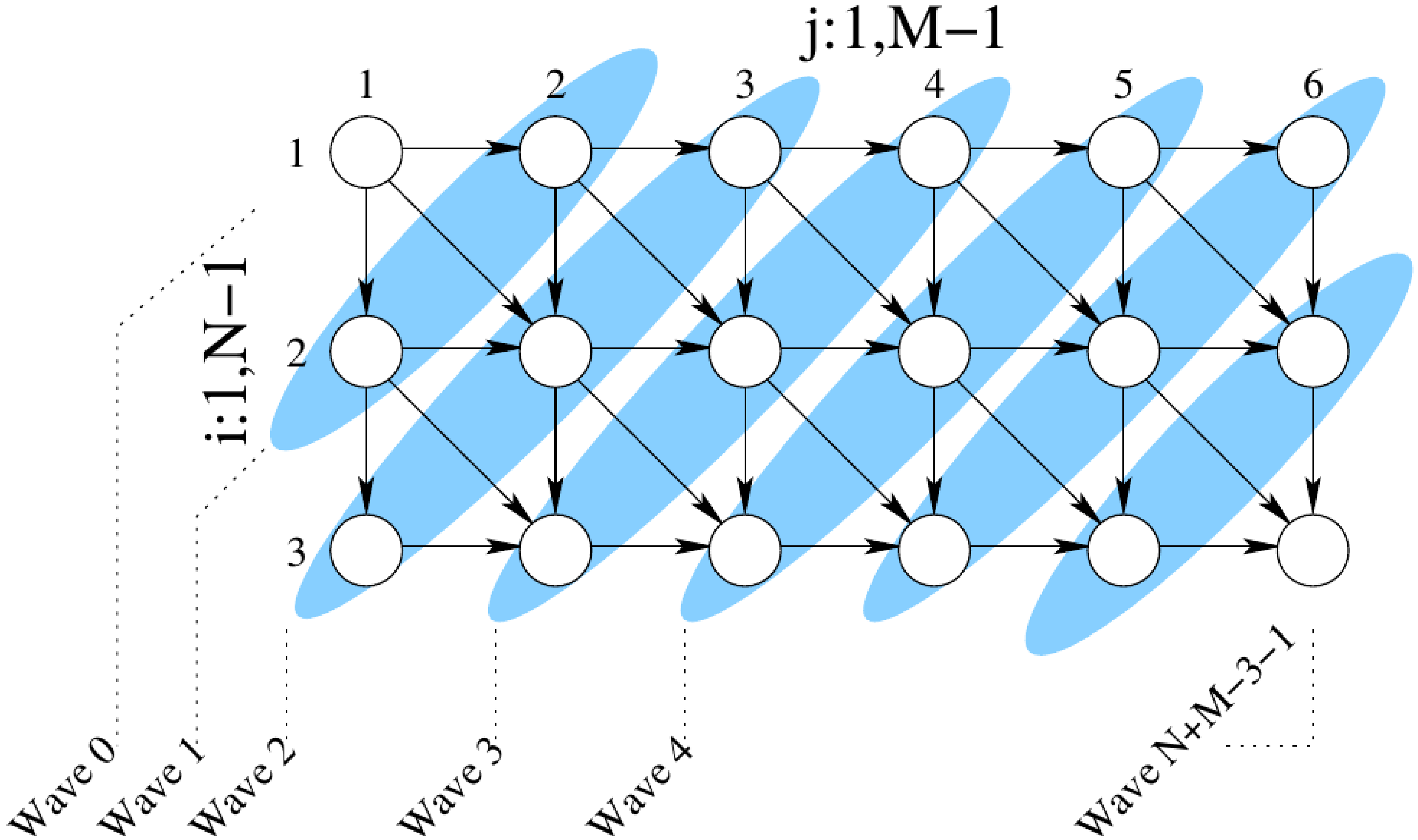
- Refactoring refers to rewriting of the loop(s) so that parallelism can be exposed.
- The ISDG for the following example:

```
for (int i = 1; i < N; i++)
  for (int j = 1; j < M; j++)
    data[i][j] = data[i - 1][j] + data[i][j - 1] + data[i - 1][↩
        j - 1];   // S1
```

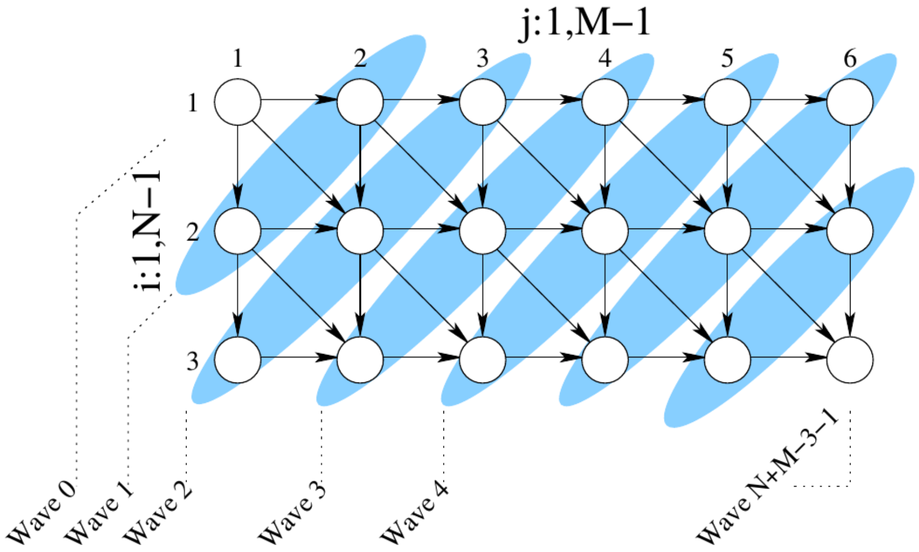


Flow Dependencies: Refactoring (2)

- Diagonal sets can be executed in parallel (no edges/dependencies between nodes in the same diagonal set):



Flow Dependencies: Refactoring (3)



```
for(wave=0 wave<NumWaves; wave++) {
  diag=F(wave);
  #pragma omp parallel for
  for(k=0; k<diag; k++) {
     int i = get_i(diag, k);
     int j = get_j(diag, k);
     data[i][j] = data[i-1][j] + data[i][j-1] + data[i-1][j-1];
   }
}
```

(Intuition, full code on the book)

The execution in waves requires a change of loop variables from the original *i* and *j*.

Data Dependency Resolution

6 techniques:

1. reduction/induction variable fix
2. Loop skewing
3. Partial parallelization
4. Refactoring
5. Fissioning
6. Algorithm change

Flow Dependencies : Fissioning

Fissioning means breaking the loop apart into a sequential and a parallelizable part.
Example:

```
s = b[ 0 ];
for (int i = 1; i < N; i++)
{
    a[ i ] = a [ i ] + a[ i - 1 ];   // S1
    s = s + b[ i ];
}
```

```
// sequential part
for (int i = 1; i < N; i++)
    a[ i ] = a [ i ] + a[ i - 1 ];

// parallel part
s = b[ 0 ];
#pragma omp parallel for reduction(+ : s)
for (int i = 1; i < N; i++)
    s = s + b[ i ];
```

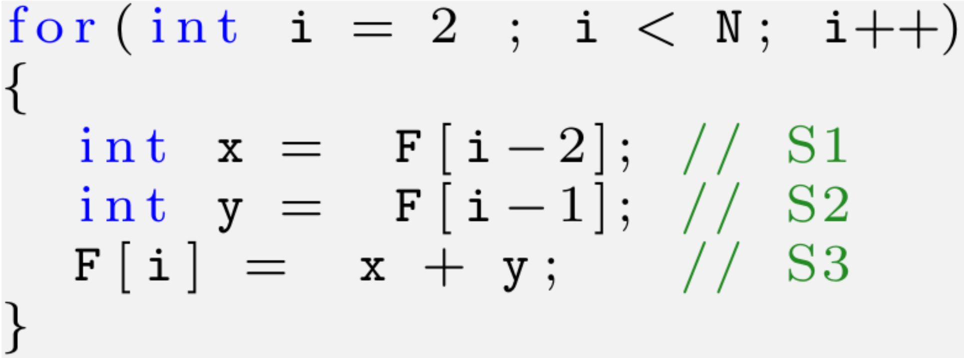
Data Dependency Resolution

6 techniques:

1. reduction/induction variable fix
2. Loop skewing
3. Partial parallelization
4. Refactoring
5. Fissioning
6. Algorithm change

Flow Dependencies: Algorithm Change

- If everything else fails, switching the algorithm maybe the answer.
- For example, the Fibonacci sequence:



```
for(int i = 2 ; i < N; i++)
{
  int x =  F[i-2]; // S1
  int y =  F[i-1]; // S2
  F[i] =  x + y;   // S3
}
```

can be parallelized via Binet's formula:

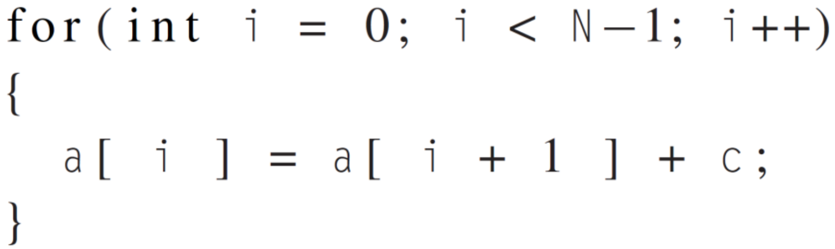
$$F_n = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}}$$

Questions?

Antidependence Removal (WAR)

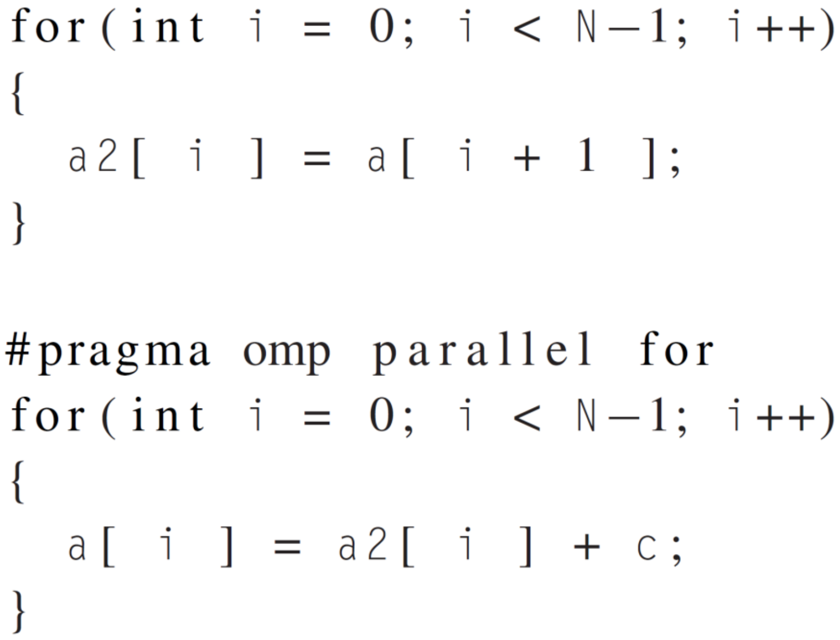
Antidependence Removal (WAR)

- E.g.:



```
for(int i = 0; i < N-1; i++)
{
  a[ i ] = a[ i + 1 ] + c;
}
```

- Simple solution, make a copy of a before starting to modify it:



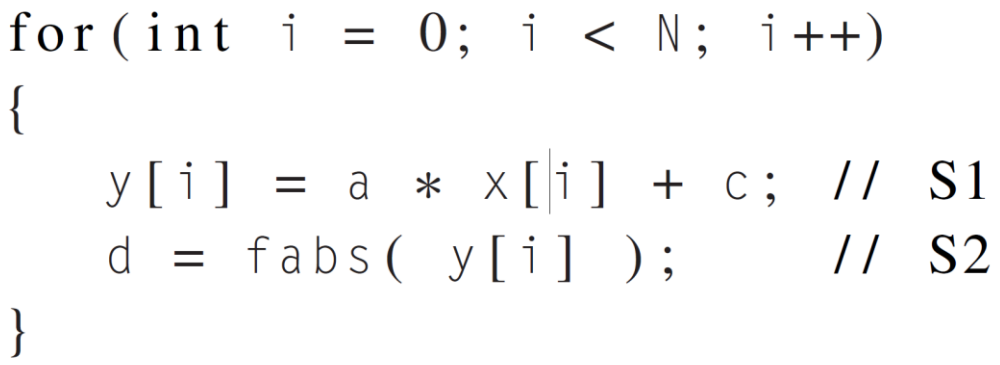
```
for(int i = 0; i < N-1; i++)
{
  a2[ i ] = a[ i + 1 ];
}

#pragma omp parallel for
for(int i = 0; i < N-1; i++)
{
  a[ i ] = a2[ i ] + c;
}
```

- ATTENTION: Space and time tradeoffs must be carefully evaluated!

Output Dependence Removal (WAW)

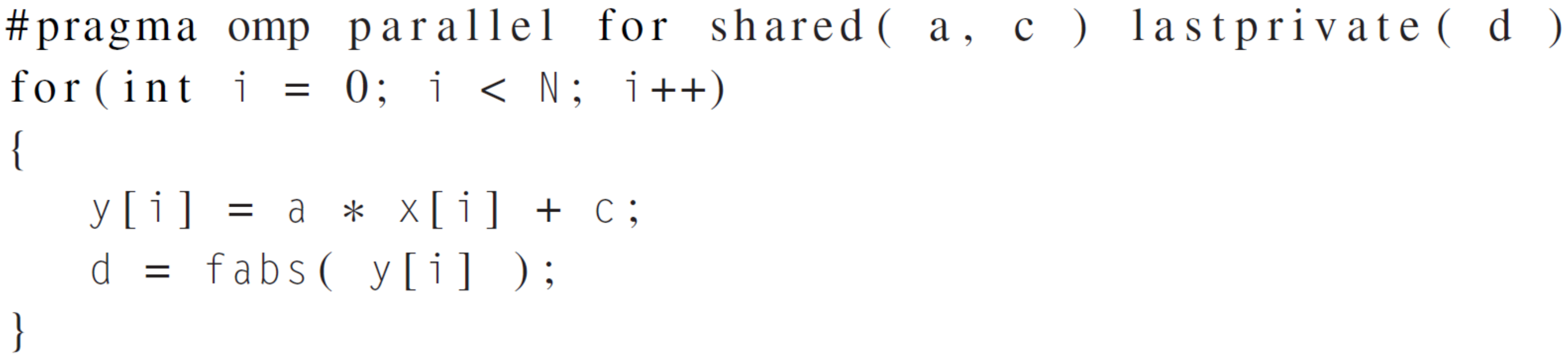
Output Dependence Removal (WAW)

- E.g.:



```
for(int i = 0; i < N; i++)
{
    y[i] = a * x[i] + c;  // S1
    d = fabs( y[i] );     // S2
}
```

- Where is the WAW dependency?
  - On variable d.
- How to guarantee that at the end of the execution the computed d is the one computed in the last iteration?



```
#pragma omp parallel for shared( a, c ) lastprivate( d )
for(int i = 0; i < N; i++)
{
    y[i] = a * x[i] + c;
    d = fabs( y[i] );
}
```

Questions?

OPIS

OPIS

OPIS Code: B76Z7JSO

Instructions: