

# Programming OpenMP

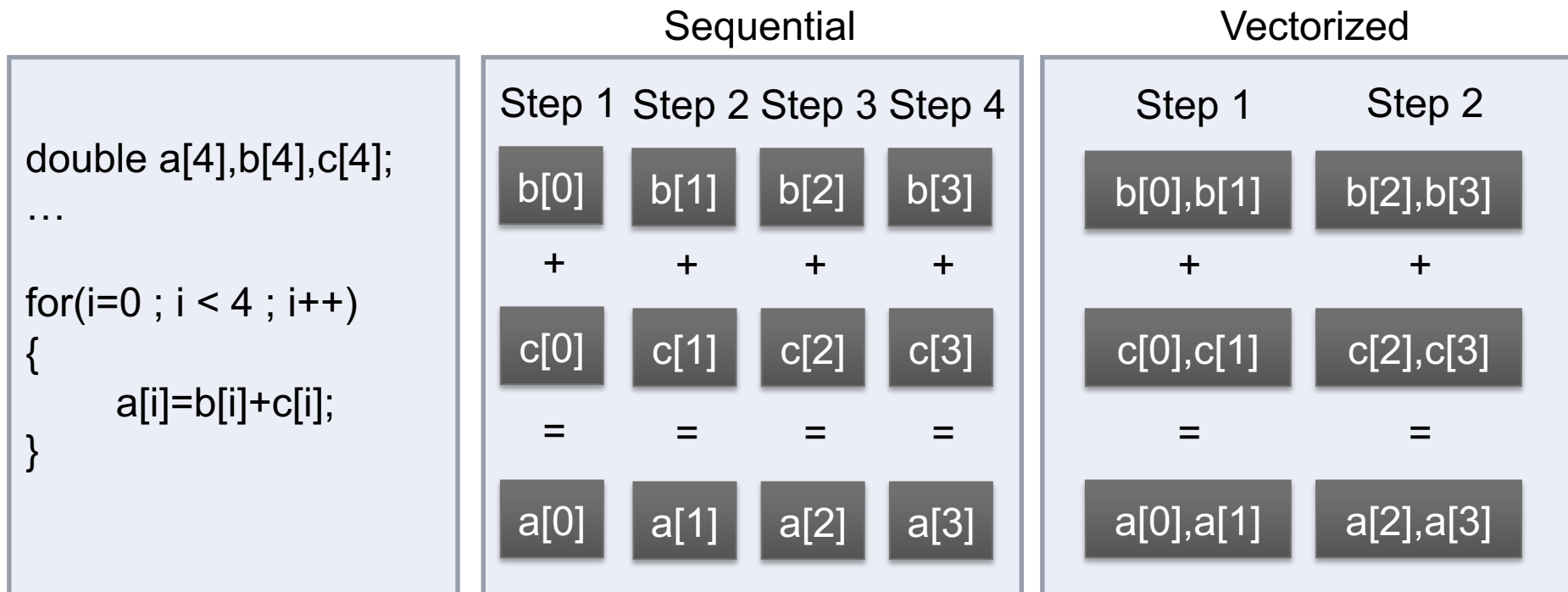
## *Vectorization (SIMD)*

Tim Cramer



## ■ SIMD = Single Instruction Multiple Data

- Special hardware instructions to operate on multiple data points at once
- Instructions work on vector registers
- Vector length is hardware dependent



## ■ Vector lengths on Intel architectures

→ 128 bit: SSE = Streaming SIMD Extensions



→ 256 bit: AVX = Advanced Vector Extensions

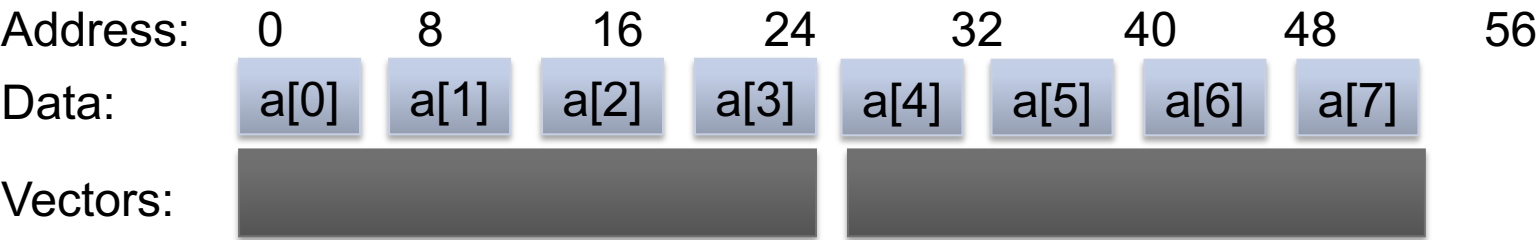


→ 512 bit: AVX-512

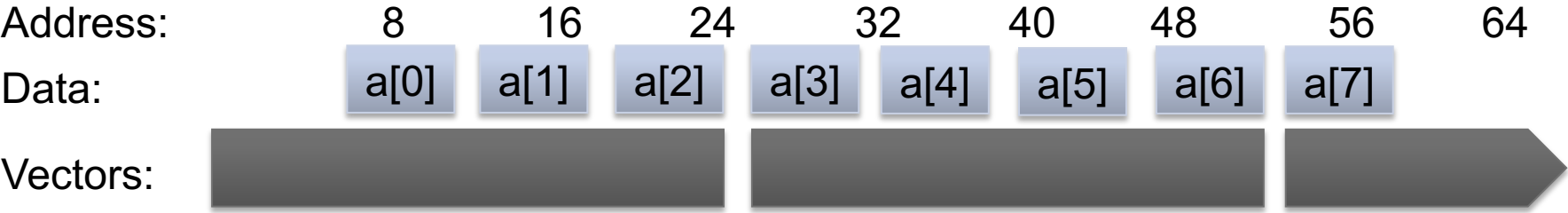


■ Vectorization works best on aligned data structures.

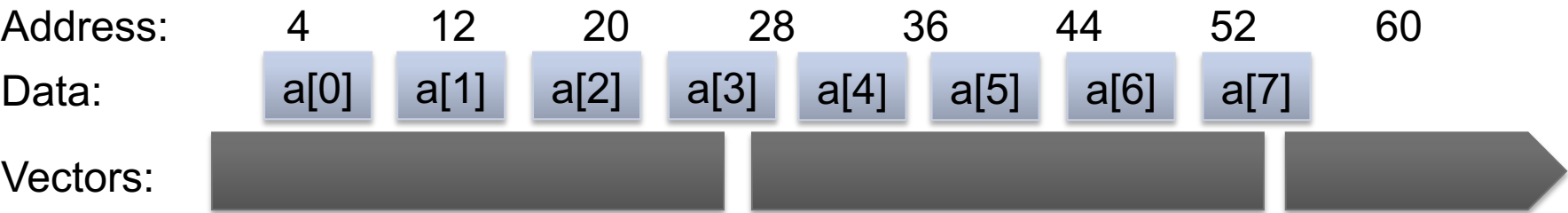
**Good alignment**



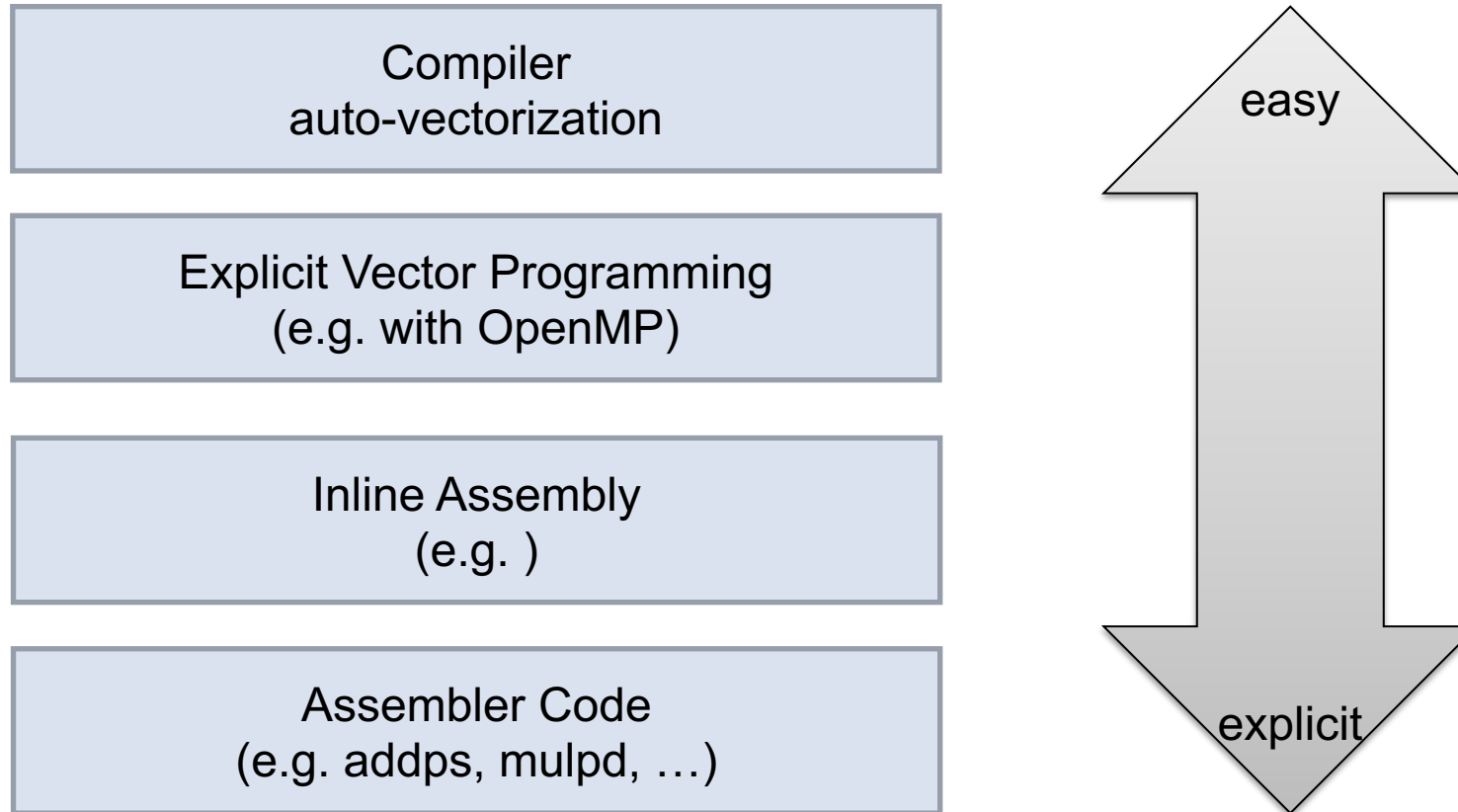
**Bad alignment**



**Very bad alignment**



# Approaches to Vectorization



# Data Dependencies

- Suppose two statements S1 and S2
- S2 depends on S1, iff S1 must execute before S2
  - Control-flow dependence
  - Data dependence
  - Dependencies can be carried over between loop iterations
- Important flavors of data dependencies

## FLOW

s1: a = 40

b = 21

s2: c = a + 2



## ANTI

b = 40

s1: a = b + 1

s2: b = 21



# Loop-Carried Dependencies

- Dependencies may occur across loop iterations

→ Loop-carried dependency

- The following code contains such a dependency:

```
void lcd_ex(float* a, float* b, size_t n, float c1, float c2)
{
    size_t i;
    for (i = 0; i < n; i++) {
        a[i] = c1 * a[i + 17] + c2 * b[i];
    }
}
```

Loop-carried dependency for  $a[i]$  and  $a[i+17]$ ; distance is 17.

- Some iterations of the loop have to complete before the next iteration can run

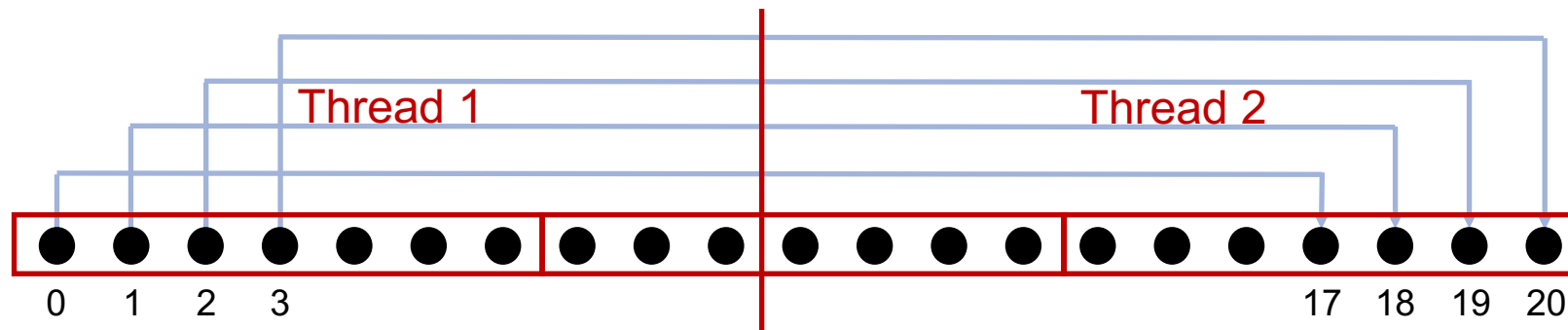
→ Simple trick: Can you reverse the loop w/o getting wrong results?



# Loop-carried Dependencies

## ■ Can we parallelize or vectorize the loop?

```
void lcd_ex(float* a, float* b, size_t n, float c1, float c2) {  
    for (int i = 0; i < n; i++) {  
        a[i] = c1 * a[i + 17] + c2 * b[i];  
    }  
}
```



- Parallelization: no  
(except for very specific loop schedules)
- Vectorization: yes  
(iff vector length is shorter than any distance of any dependency)

# The OpenMP SIMD constructs

## The SIMD construct

- The SIMD construct enables the execution of multiple iterations of the associated loops concurrently by means of SIMD instructions.

C/C++:

```
#pragma omp simd [clause(s)]  
for-loops
```

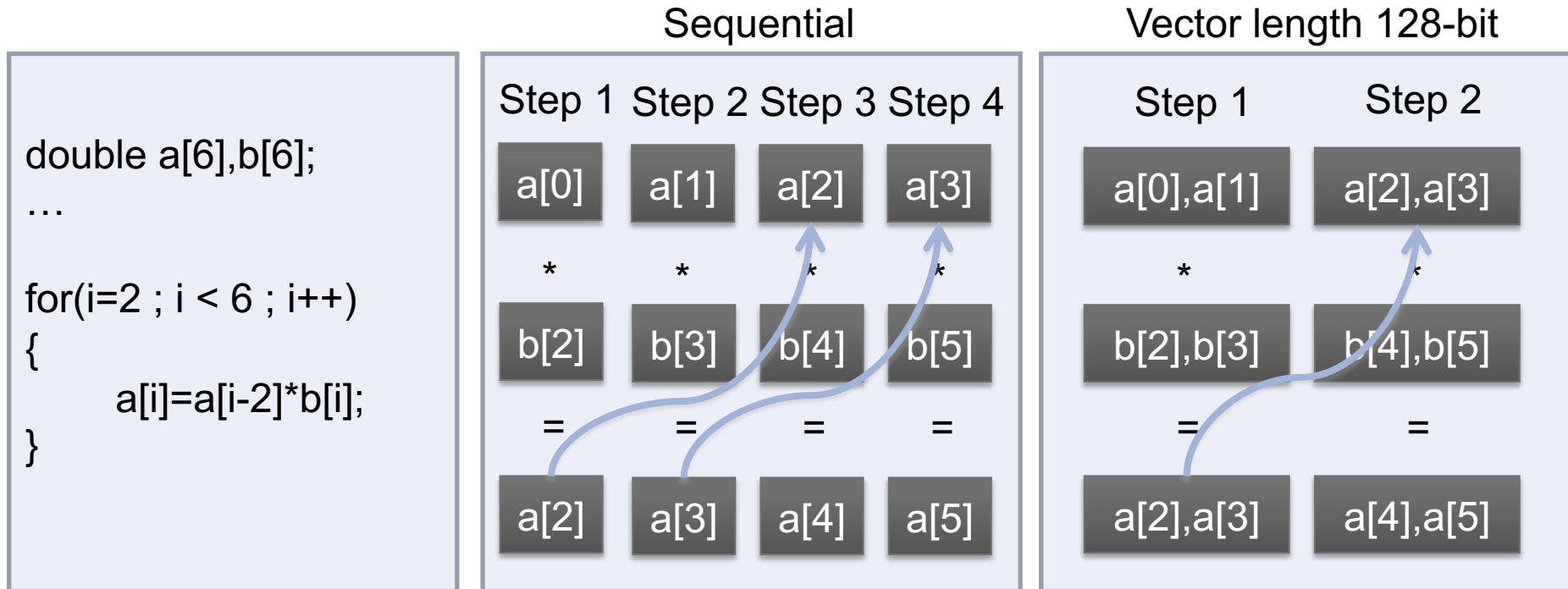
Fortran:

```
!$omp simd [clause(s)]  
do-loops  
[!$omp end simd]
```

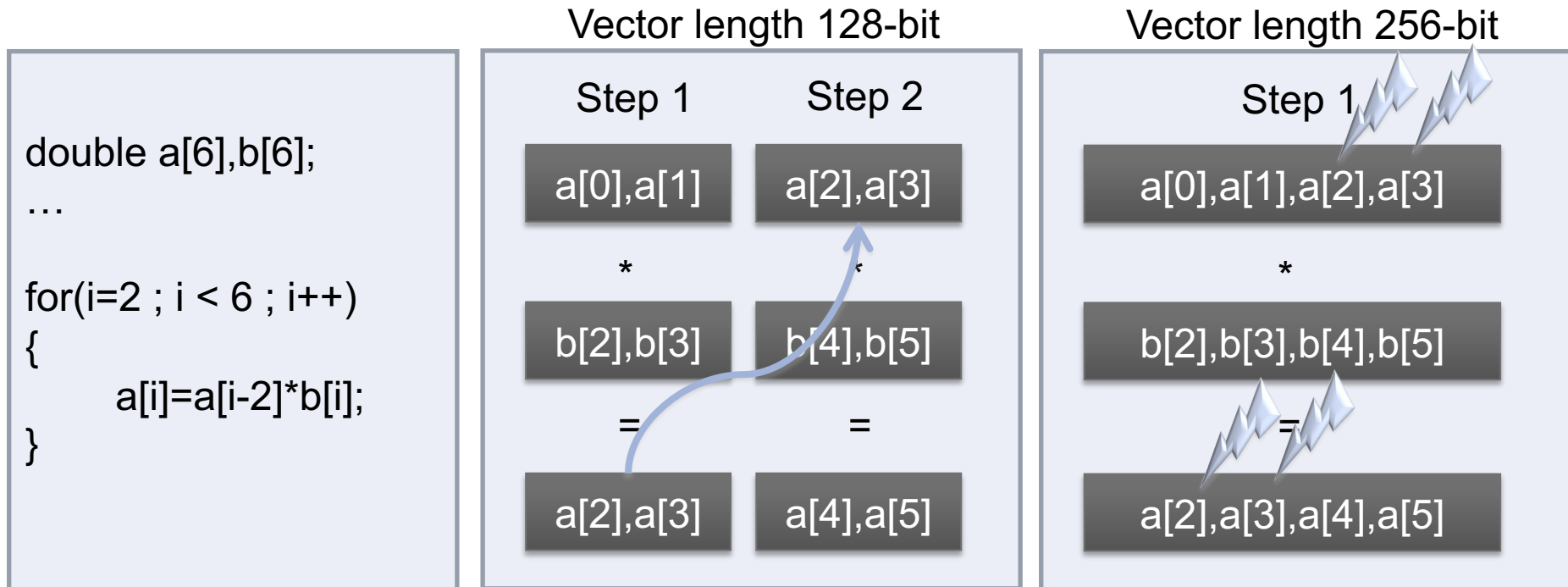
- where clauses are:

- linear(*list[:linear-step]*), a variable increases linearly in every loop iteration
- aligned(*list[:alignment]*), specifies that data is aligned
- private(*list*), as usual
- lastprivate(*list*) , as usual
- reduction(*reduction-identifier:list*) , as usual
- collapse(*n*), collapse loops first, and then apply SIMD instructions

- The safelen clause allows to specify a distance of loop iterations where no dependencies occur.



- The safelen clause allows to specify a distance of loop iterations where no dependencies occur.



- Any vector length smaller than or equal to the length specified by safelen can be chosen for vectorization.
- In contrast to parallel for/do loops the iterations are executed in a specified order.

## The loop SIMD construct

- The loop SIMD construct specifies a loop that can be executed in parallel by all threads and in SIMD fashion on each thread.

C/C++:

```
#pragma omp for simd [clause(s)]  
for-loops
```

Fortran:

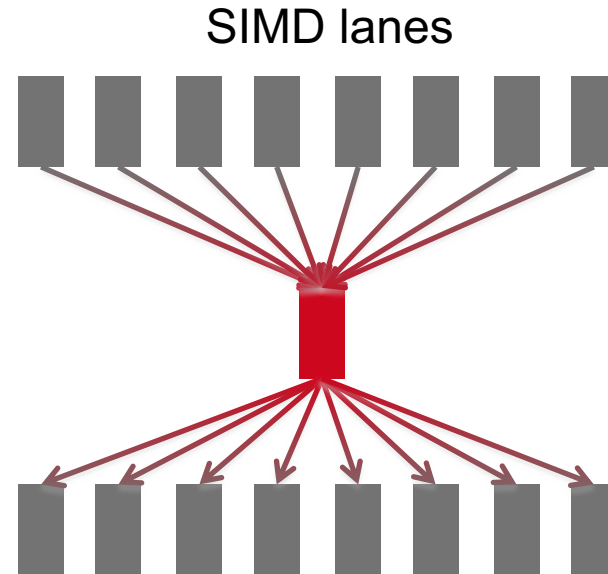
```
!$omp do simd [clause(s)]  
do-loops  
[!$omp end do simd [nowait]]
```

- Loop iterations are first distributed across threads, then each chunk is handled as a SIMD loop.
- Clauses:
  - All clauses from the *loop*- or SIMD-construct are allowed
  - Clauses which are allowed for both constructs are applied twice, once for the threads and once for the SIMDization.

## The declare SIMD construct

- **Function calls in SIMD-loops can lead to bottlenecks, because functions need to be executed serially.**

```
for(i=0 ; i < N ; i++)  
{  
    a[i]=b[i]+c[i];  
    d[i]=sin(a[i]);  
    e[i]=5*d[i];  
}
```



Solutions:

- avoid or inline functions
- create functions which work on vectors instead of scalars

## The declare SIMD construct

- Enables the creation of multiple versions of a function or subroutine where one or more versions can process multiple arguments using SIMD instructions.

C/C++:

```
#pragma omp declare simd [clause(s)]  
[#pragma omp declare simd [clause(s)]]  
  function definition / declaration
```

Fortran:

```
!$omp declare simd (proc_name)[clause(s)]
```

- **where clauses are:**

- `simdlen(length)`, the number of arguments to process simultaneously
- `linear(list[:linear-step])`, a variable increases linearly in every loop iteration
- `aligned(argument-list[:alignment])`, specifies that data is aligned
- `uniform(argument-list)`, arguments have an invariant value
- `inbranch / notinbranch`, function is always/never called from within a conditional statement



File: f.c

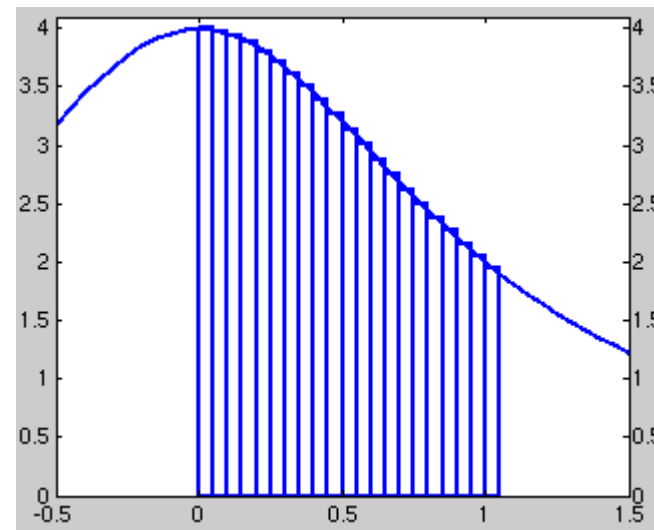
```
#pragma omp declare simd
double f(double x)
{
    return (4.0 / (1.0 + x*x));
}
```

File: pi.c

```
#pragma omp declare simd
double f(double x);
...
#pragma omp simd linear(i) private(fX)
reduction(+:fSum)
for (i = 0; i < n; i++)
{
    fX = fH * ((double)i + 0.5);
    fSum += f(fX);
}
return fH * fSum;
```

Calculating Pi with  
numerical integration  
of:

$$\pi = \int_0^1 \frac{4}{1+x^2}$$



## Example: Pi

### ■ Runtime of the benchmark on:

- Westmere CPU with SSE (128-bit vectors)
- Intel Xeon Phi with AVX-512 (512-bit vectors)

	Runtime Westmere	Speedup Westmere	Runtime Xeon Phi	Speedup Xeon Phi
non vectorized	1.44 sec	1	16.25 sec	1
vectorized	0.72 sec	2	1.82 sec	8.9

**Note:** Speedup for memory bound applications might be lower on both systems.