

Advanced OpenMP Topics

Christian Terboven, Dirk Schmidl

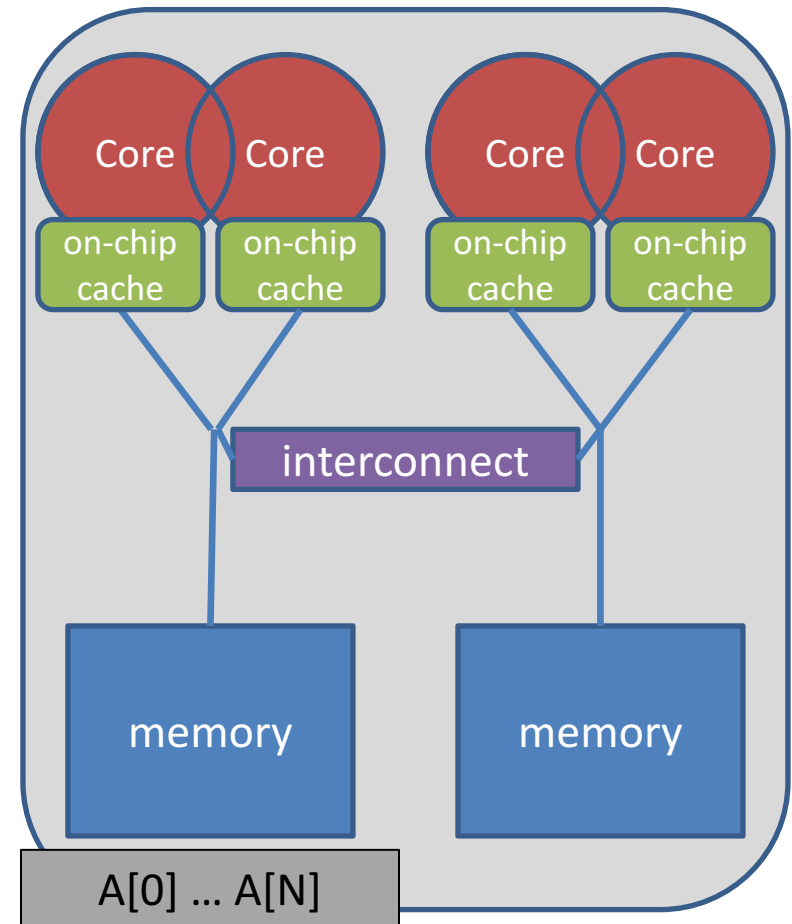
IT Center, RWTH Aachen University
{terboven,schmidl}@itc.rwth-aachen.de

NUMA Architectures

How To Distribute The Data ?

```
double* A;  
A = (double*)  
    malloc(N * sizeof(double));
```

```
for (int i = 0; i < N; i++) {  
    A[i] = 0.0;  
}
```



- **Important aspect on cc-NUMA systems**

- If not optimal, longer memory access times and hotspots

- **OpenMP does not provide support for cc-NUMA**

- **Placement comes from the Operating System**

- This is therefore Operating System dependent

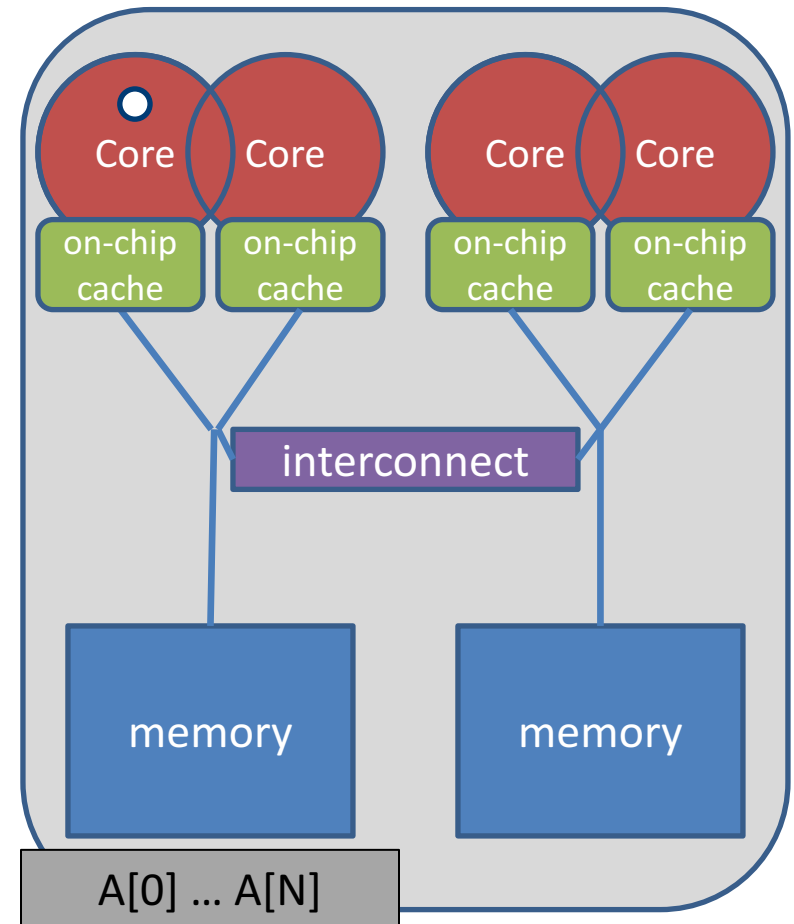
- **Windows, Linux and Solaris all use the “First Touch” placement policy by default**

- May be possible to override default (check the docs)

- Serial code: all array elements are allocated in the memory of the NUMA node containing the core executing this thread

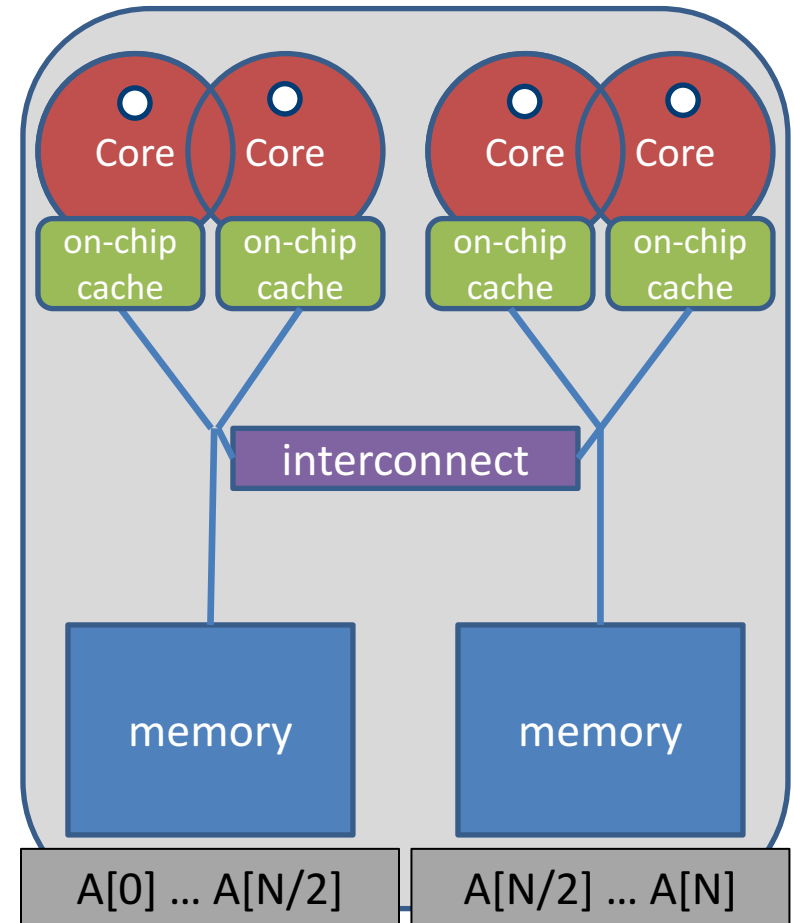
```
double* A;  
A = (double*)  
    malloc(N * sizeof(double));
```

```
for (int i = 0; i < N; i++) {  
    A[i] = 0.0;  
}
```



- **First Touch w/ parallel code: all array elements are allocated in the memory of the NUMA node containing the core executing the thread initializing the respective partition**

```
double* A;  
A = (double*)  
    malloc(N * sizeof(double));  
  
omp_set_num_threads(4);  
  
#pragma omp parallel for  
for (int i = 0; i < N; i++) {  
    A[i] = 0.0;  
}
```



- **Before you design a strategy for thread binding, you should have a basic understanding of the system topology. Please use one of the following options on a target machine:**

- Intel MPI's `cpuinfo` tool

- `module switch openmpi intelmpi`

- `cpuinfo`

- Delivers information about the number of sockets (= packages) and the mapping of processor ids used by the operating system to cpu cores.

- `hwloc`'tools

- `lstopo` (command line: `hwloc-ls`)

- Displays a graphical representation of the system topology, separated into NUMA nodes, along with the mapping of processor ids used by the operating system to cpu cores and additional info on caches.

- **Selecting the „right“ binding strategy depends not only on the topology, but also on the characteristics of your application.**
 - Putting threads far apart, i.e. on different sockets
 - May improve the aggregated memory bandwidth available to your application
 - May improve the combined cache size available to your application
 - May decrease performance of synchronization constructs
 - Putting threads close together, i.e. on two adjacent cores which possibly shared some caches
 - May improve performance of synchronization constructs
 - May decrease the available memory bandwidth and cache size
- **If you are unsure, just try a few options and then select the best one.**



■ Define OpenMP Places

- set of OpenMP threads running on one or more processors
- can be defined by the user, i.e. `OMP_PLACES=cores`

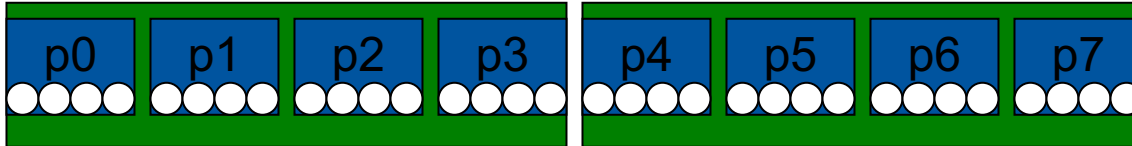
■ Define a set of OpenMP Thread Affinity Policies

- SPREAD: spread OpenMP threads evenly among the places
- CLOSE: pack OpenMP threads near master thread
- MASTER: collocate OpenMP thread with master thread

■ Goals

- user has a way to specify where to execute OpenMP threads for
- locality between OpenMP threads / less false sharing / memory bandwidth

■ Assume the following machine:



→ 2 sockets, 4 cores per socket, 4 hyper-threads per core

■ Abstract names for OMP_PLACES:

→ threads: Each place corresponds to a single hardware thread on the target machine.

→ cores: Each place corresponds to a single core (having one or more hardware threads) on the target machine.

→ sockets: Each place corresponds to a single socket (consisting of one or more cores) on the target machine.

Example's Objective:

→ separate cores for outer loop and near cores for inner loop

Outer Parallel Region: `proc_bind(spread)`, Inner: `proc_bind(close)`

→ spread creates partition, compact binds threads within respective partition

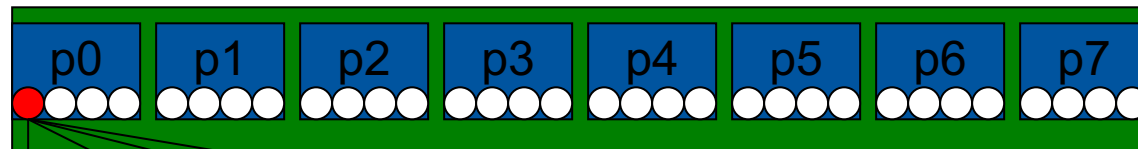
```
OMP_PLACES=(0,1,2,3), (4,5,6,7), ... = (0-3):8:4 = cores
```

```
#pragma omp parallel proc_bind(spread) num_threads(4)
```

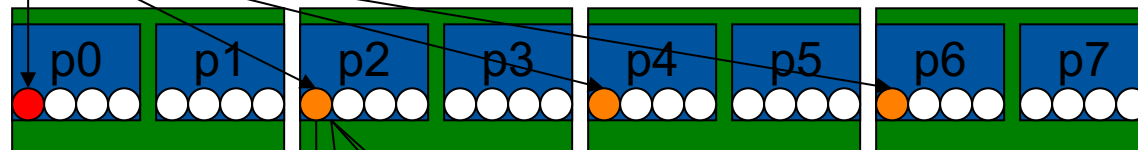
```
#pragma omp parallel proc_bind(close) num_threads(4)
```

Example

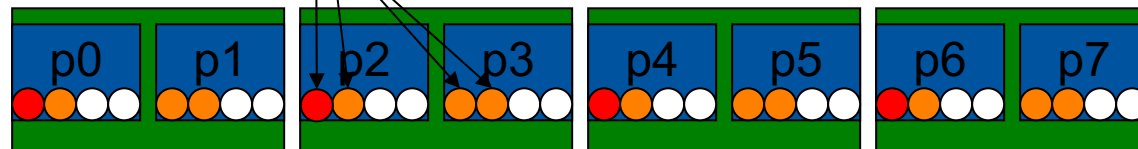
→ initial



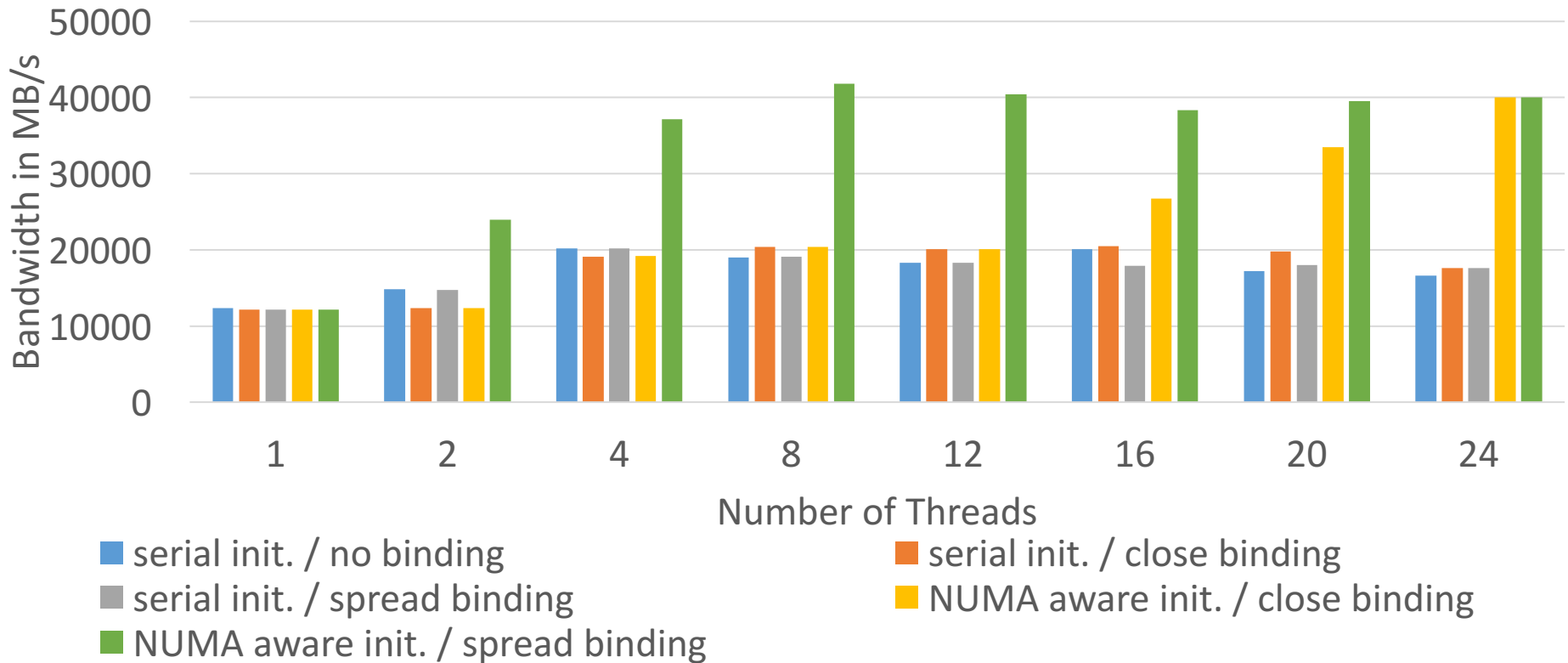
→ spread 4



→ close 4



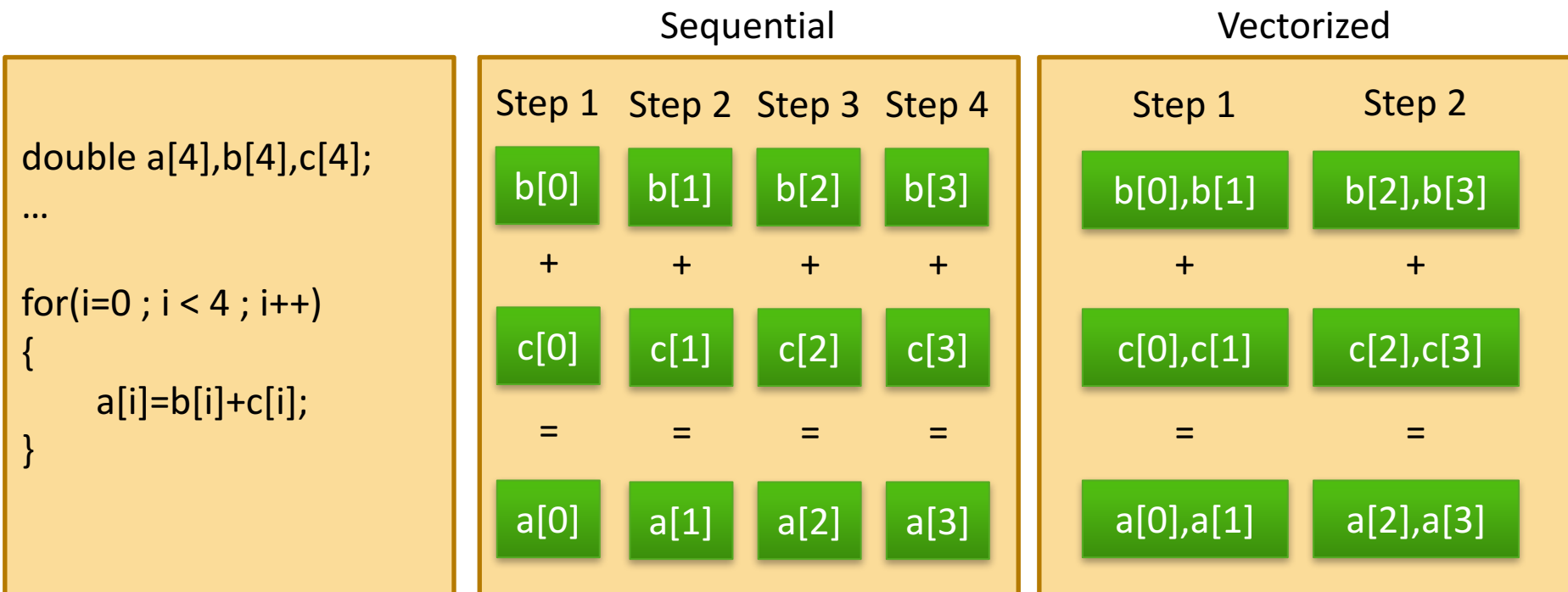
- **Performance of OpenMP-parallel STREAM vector assignment measured on 2-socket Intel® Xeon® X5675 („Westmere“) using Intel® Composer XE 2013 compiler with different thread binding options:**



Vectorization (SIMD)

■ 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

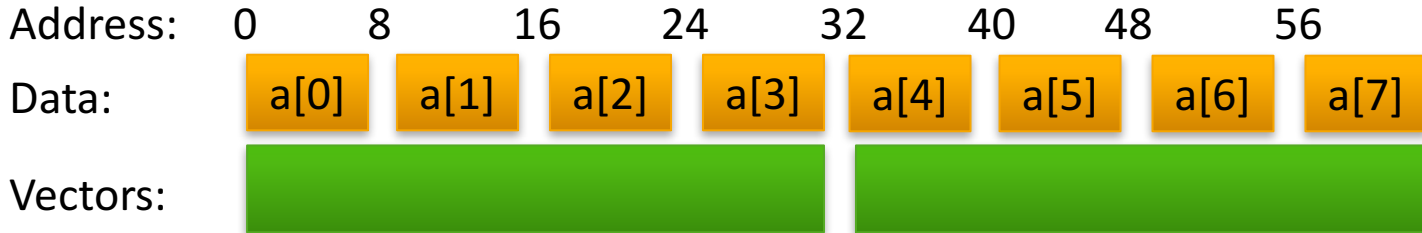


→ 512 bit: AVX-512

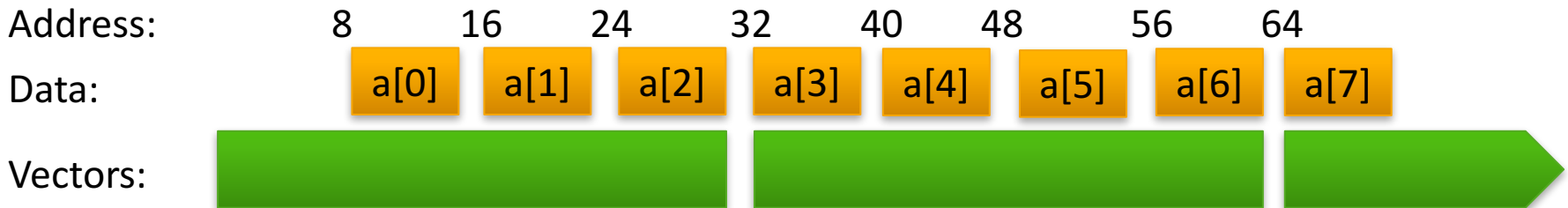


■ Vectorization works best on aligned data structures.

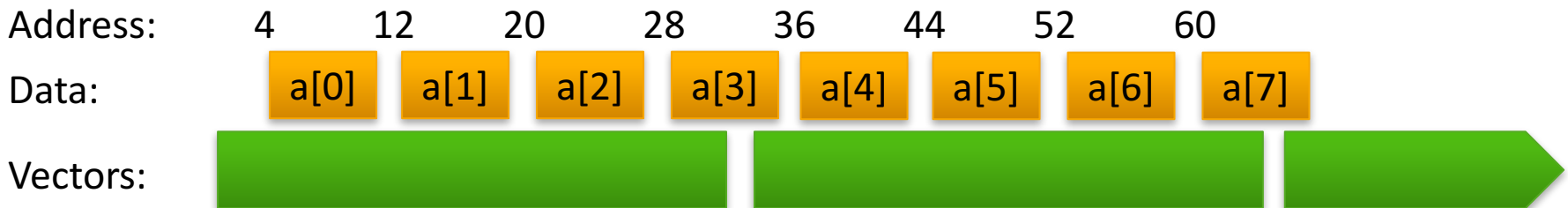
Good alignment



Bad alignment



Very bad alignment



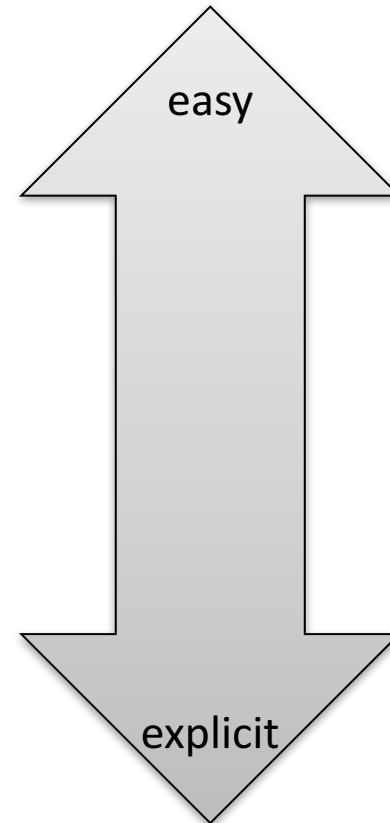
■ Ways to Vectorize

Compiler
auto-vectorization

Explicit Vector Programming
(e.g. with OpenMP)

Inline Assembly
(e.g.)

Assembler Code
(e.g. `addps`, `mulpd`, ...)



The OpenMP SIMD constructs

- 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
```

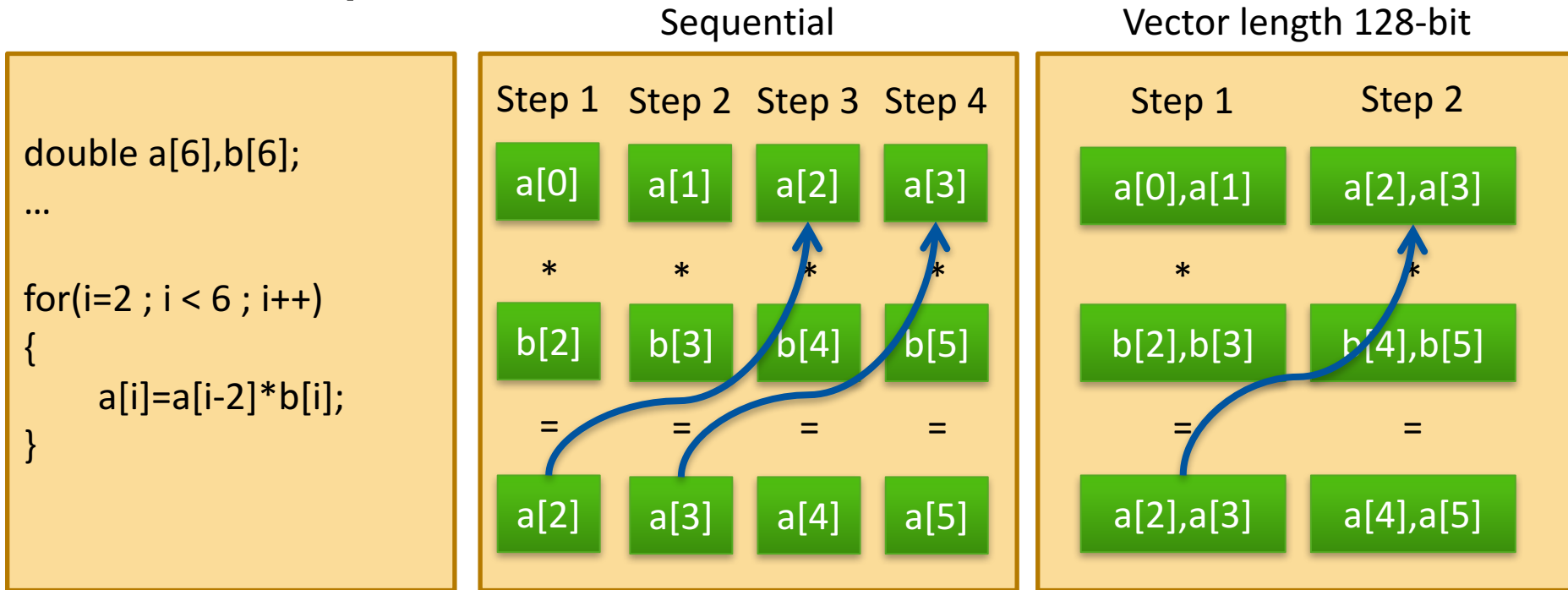
Fortran:

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

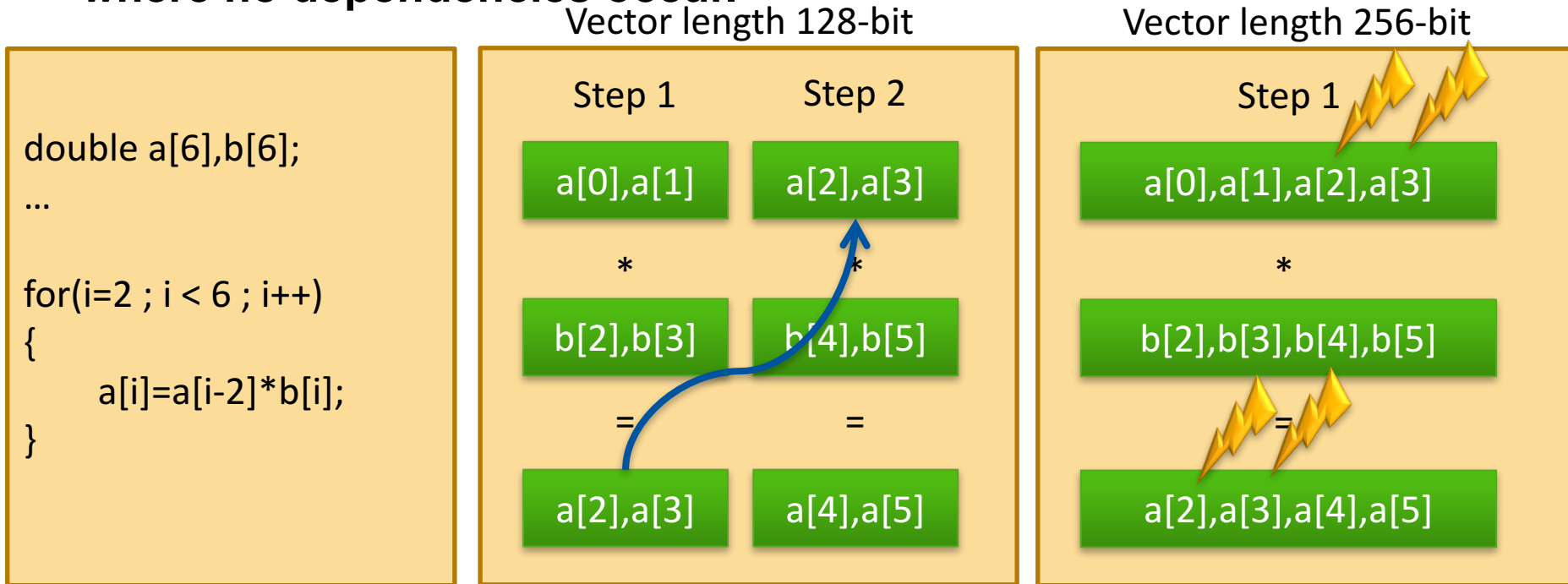
- where clauses are:

- `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 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
```

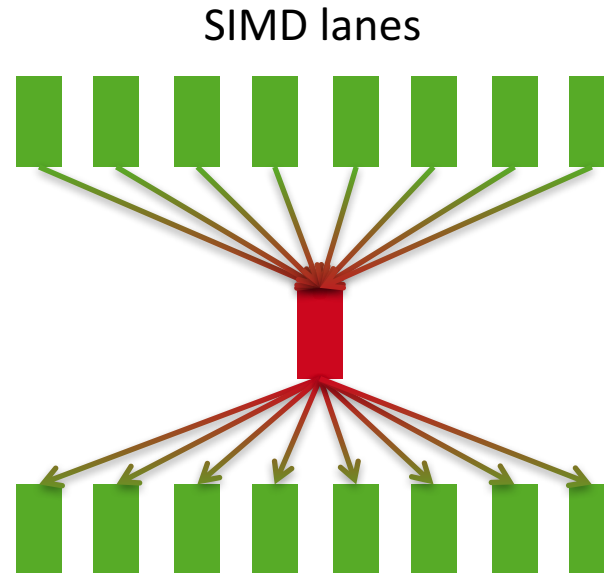
Fortran:

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

- **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

- 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

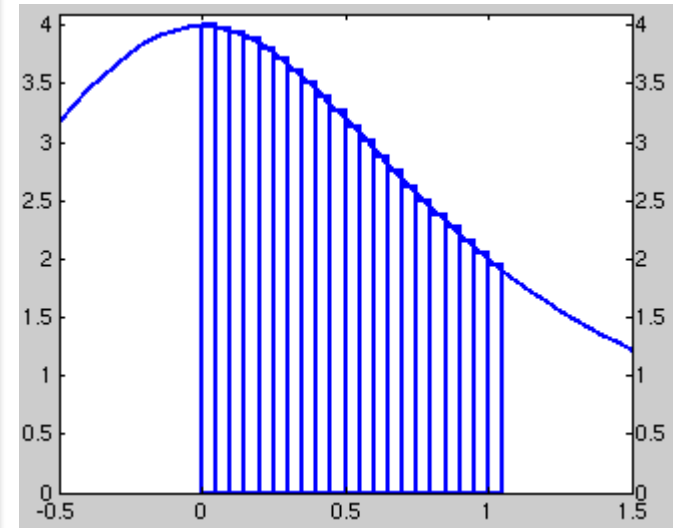
```
#pragma omp declare simd
double f(double x)
{
    return (4.0 / (1.0 + x*x));
}
```

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 1: 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.



Questions?