Further OpenMP 4.0 Features

Dirk Schmidl
IT Center, RWTH Aachen University
Member of the HPC Group
schmidl@itc.rwth-aachen.de

Christian Terboven
IT Center, RWTH Aachen University
Deputy lead of the HPC Group
terboven@itc.rwth-aachen.de

Vectorization

Vectorization

- **SIMD = S**ingle **I**nstruction **M**ultiple **D**ata
  - → Special hardware instructions to operate on multiple data points at once
  - → Instructions work on vector registers
  - → Vector length is hardware dependent

```
double a[4],b[4],c[4];
…

for(i=0 ; i < 4 ; i++)
{
	a[i]=b[i]+c[i];
}
```

Sequential

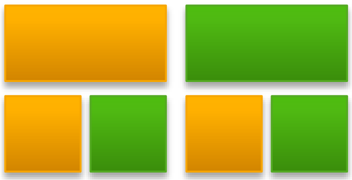
Step 1	Step 2	Step 3	Step 4
b[0]	b[1]	b[2]	b[3]
+	+	+	+
c[0]	c[1]	c[2]	c[3]
=	=	=	=
a[0]	a[1]	a[2]	a[3]

Vectorized

Step 1	Step 2
b[0],b[1]	b[2],b[3]
+	+
c[0],c[1]	c[2],c[3]
=	=
a[0],a[1]	a[2],a[3]

Vector lengths on Intel architectures

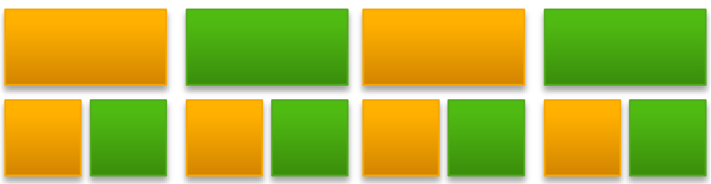
- → 128 bit: SSE = Streaming SIMD Extensions



2 x double

4 x float

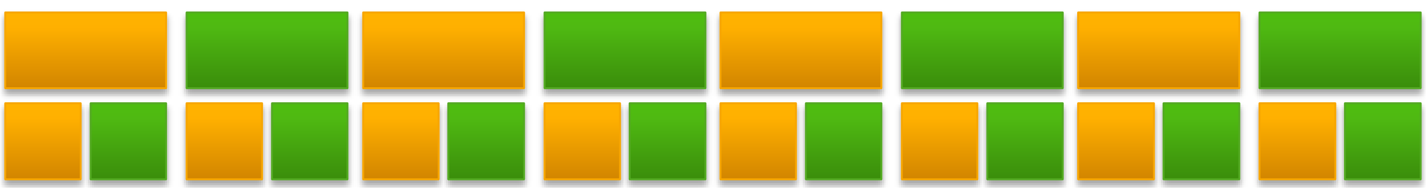
- → 256 bit: AVX = Advanced Vector Extensions



4 x double

8 x float

- → 512 bit: AVX-512

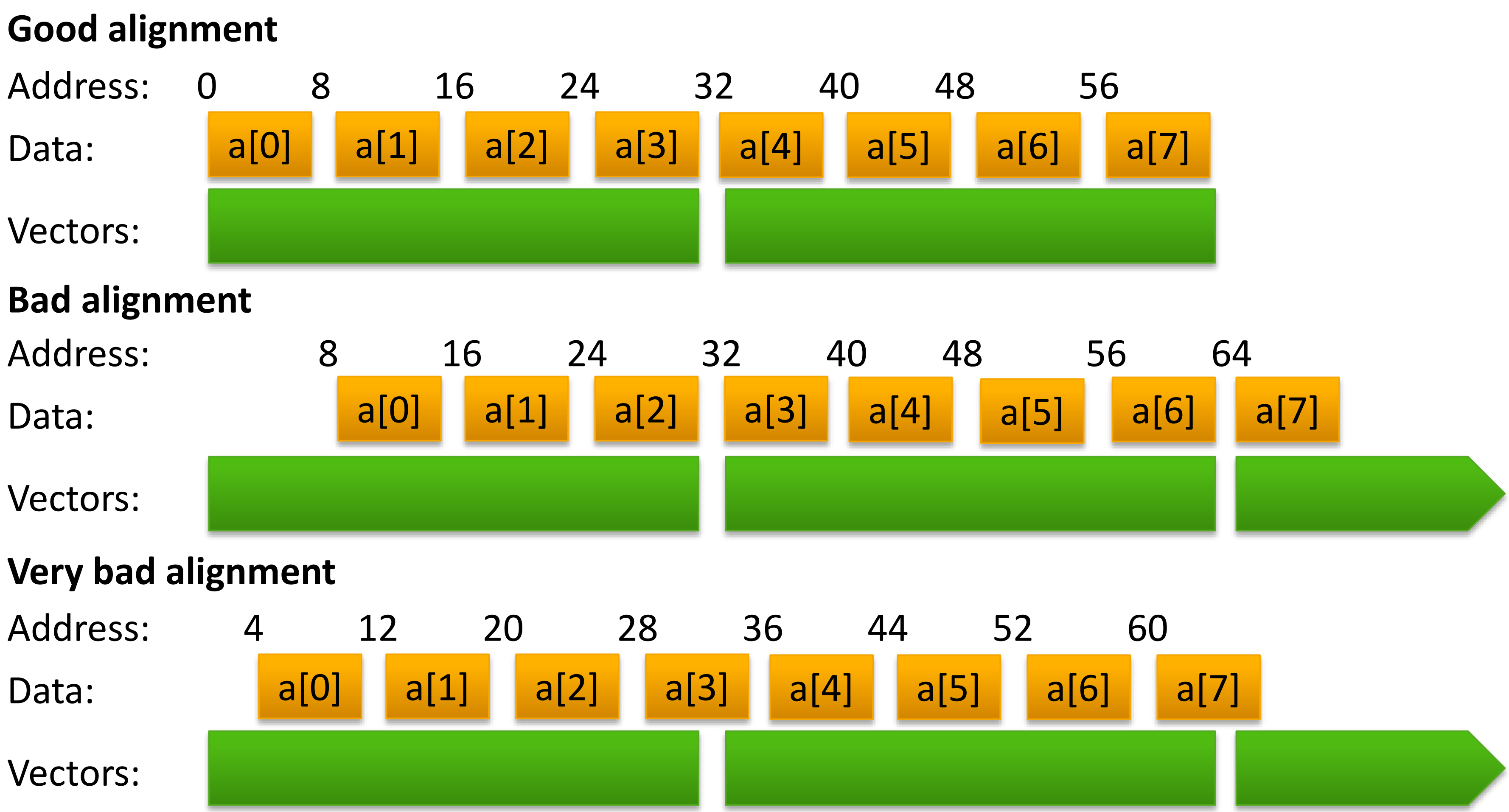


8 x double

16 x float

Data Alignment

- Vectorization works best on aligned data structures.



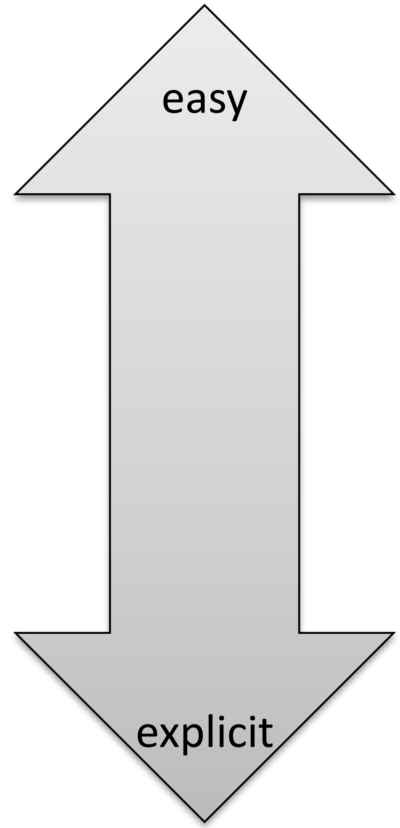
Ways to Vectorize

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

- **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]
```

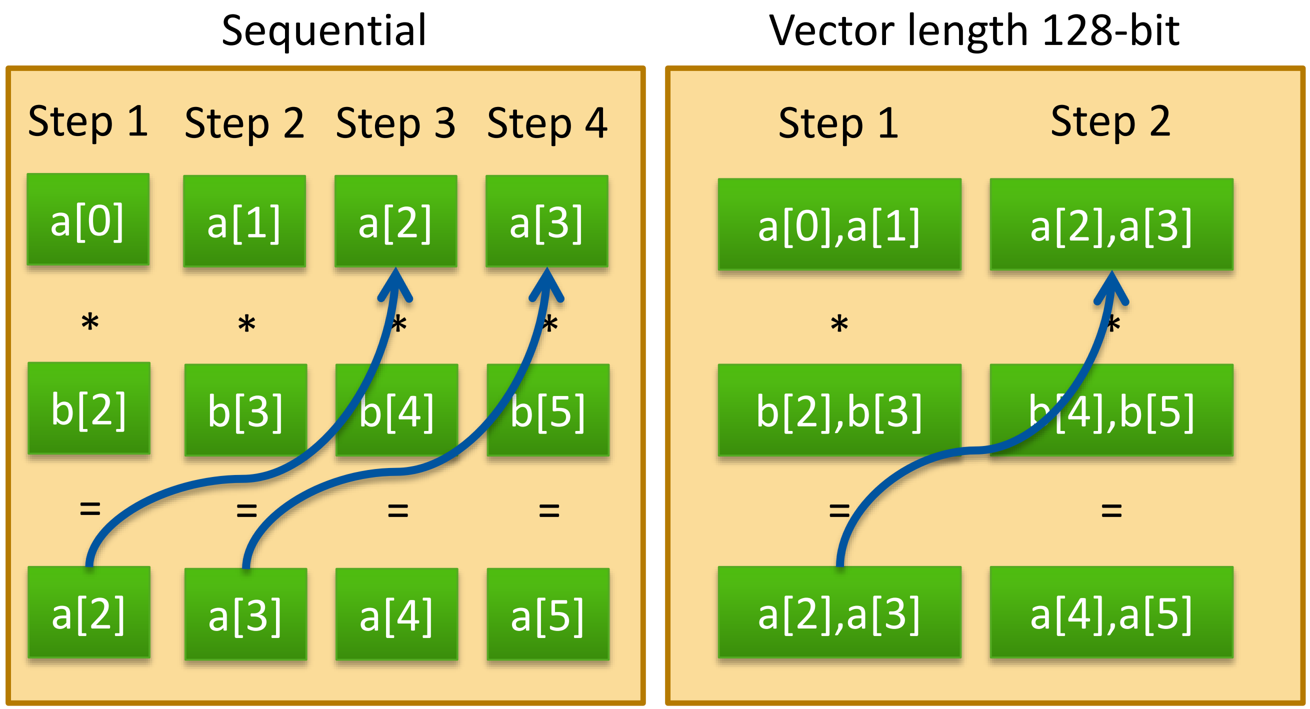
- **where clauses are:**
  - → linear(*list[:linear-step]*), a variable increases linearly in every loop iteration
  - → aligned(*list[:alignment]*), specifies that data is aligned
  - → private(*list*), as usual
  - → lastprivate(*list*) , as usual
  - → reduction(*reduction-identifier:list*) , as usual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

The SIMD construct

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

```
double a[6],b[6];
…

for(i=2 ; i < 6 ; i++)
{
      a[i]=a[i-2]*b[i];
}
```

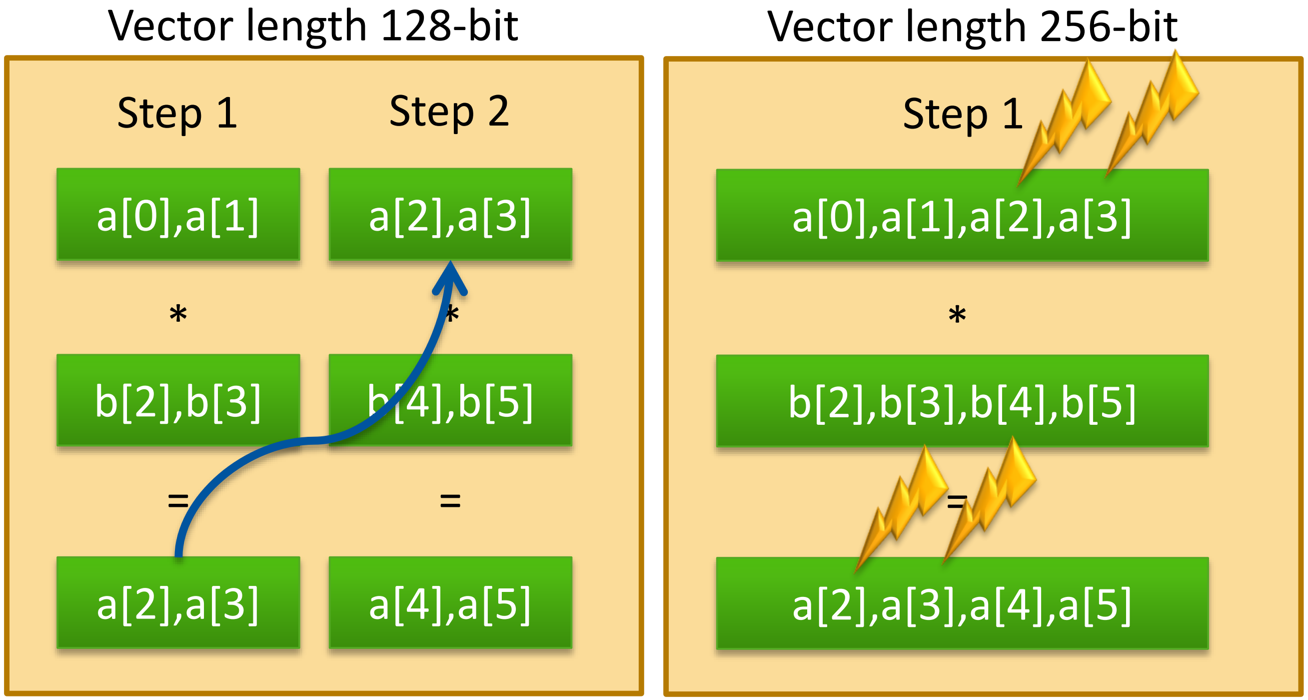


The SIMD construct

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

```
double a[6],b[6];
…

for(i=2 ; i < 6 ; i++)
{
      a[i]=a[i-2]*b[i];
}
```



- **Any vector length smaller than or equal to the length specified by safelen can be chosen for vectorizaion.**
- **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.

Dirk Schmidl, Christian Terboven | IT Center der RWTH Aachen University

The declare SIMD construct

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

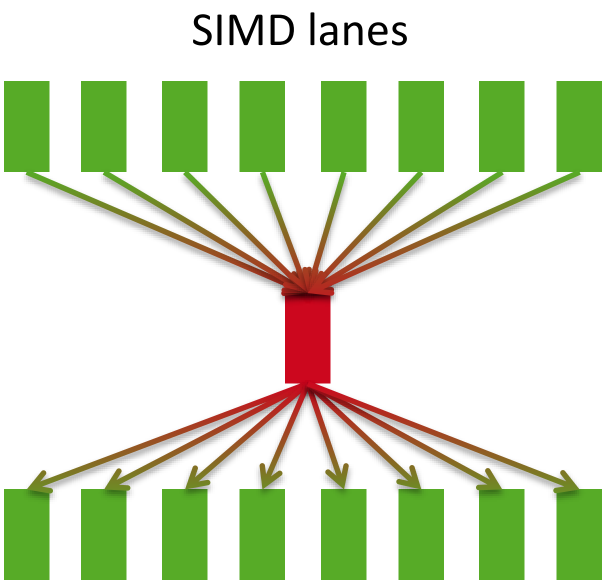
```
for(i=0 ; i < N ; i++)
{

    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 simultanously
  - → 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

PI Example Code

File: f.c

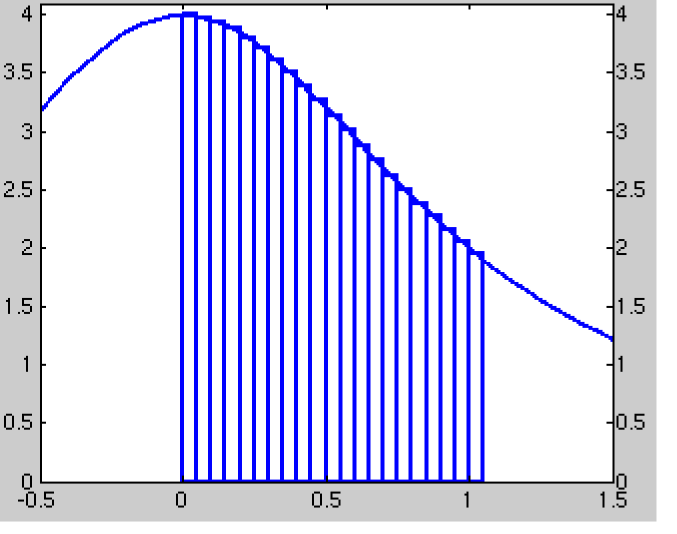
```
#pragma omp declare simd
double f(double x)
{
   return (4.0 / (1.0 + x*x));
}
```

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

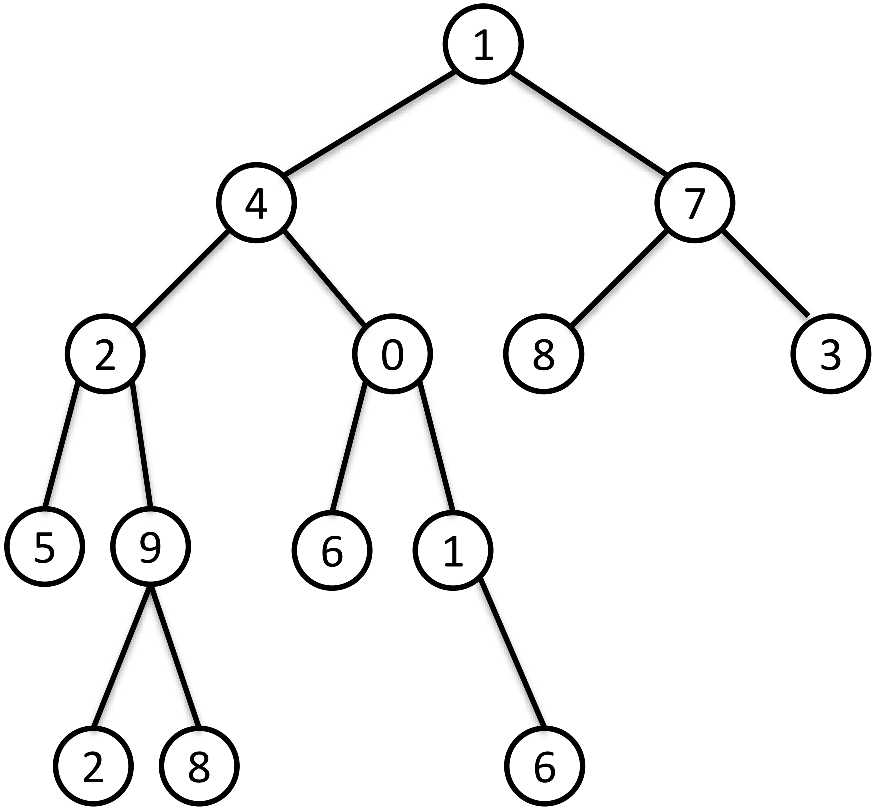
Note: Speedup for memory bound applications might be lower on both systems.

Cancellation

Cancellation

Searching in a binary tree data structure

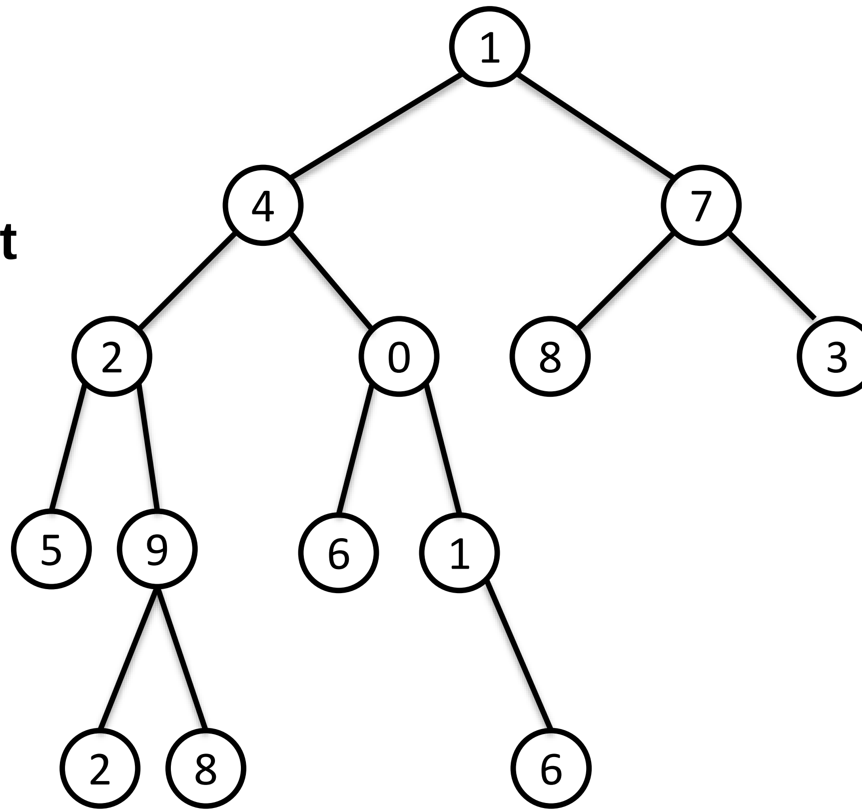
- **Tasks can be created for the left and the right subtree, leading to a task parallel search.**
- **Multiple threads can pick up the tasks and execute them.**
- **The search can be finished when one thread found the desired value.**



Cancellation

How to end the search process if one thread found the value?

- **A shared variable can be used which is set when the value is found.**
- **All threads check for this variable at every node and stop when the variable is set.**
- **A lot of flushes are needed to make sure the variable is synchronized.**
- **All tasks need to be dequeued to check the variable.**



=> This is a very inelegant and error prone solution.

Cancellation

- **Users can request cancellation of a construct**
- **threads / tasks will be canceled and execution continues after the end of the construct**

C/C++:

```
#pragma omp cancel [construct-type]
```

Fortran:

```
!$omp cancel [construct-type]
```

- **Types are: parallel, do/for, taskgroup, sections**

- **threads/ tasks stop execution at a certain point and not immediately**
- **Users can add cancellation points manually:**

C/C++:

```
#pragma omp cancellation point
```

Fortran:

```
!$omp cancellation point
```

Task Dependencies

The depend Clause

C/C++

```
#pragma omp task depend(dependency-type: list)
... structured block ...
```

- **The *task dependence* is fulfilled when the predecessor task has completed**
  - → `in` dependency-type: the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `out` or `inout` clause.
  - → `out` and `inout` dependency-type: The generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `in`, `out`, or `inout` clause.
  - → The list items in a `depend` clause may include array sections.

Concurrent Execution w/ Dep.

Degree of parallism exploitable in this concrete example:
T2 and T3 (2 tasks), T1 of next iteration has to wait for them

```
void process_in_parallel() {
   #pragma omp parallel
   #pragma omp single
   {
      int x = 1;
      ...
      for (int i = 0; i < T; ++i) {
         #pragma omp task shared(x, ...) depend(out: x)   // T1
            preprocess_some_data(...);
         #pragma omp task shared(x, ...) depend(in: x)    // T2
            do_something_with_data(...);
         #pragma omp task shared(x, ...) depend(in: x)    // T3
            do_something_independent_with_data(...);
      }
   } // end omp single, omp parallel
}
```

T1 has to be completed before T2 and T3 can be executed.

T2 and T3 can be executed in parallel.

Concurrent Execution w/ Dep.

- **The following allows for more parallelism, as there is one i per thread. Hence, two tasks my be active per thread.**

```
void process_in_parallel() {
   #pragma omp parallel
   {
      #pragma omp for
      for (int i = 0; i < T; ++i) {
         #pragma omp task depend(out: i)
            preprocess_some_data(...);
         #pragma omp task depend(in: i)
            do_something_with_data(...);
         #pragma omp task depend(in: i)
            do_something_independent_with_data(...);
      }
   } // end omp parallel
}
```

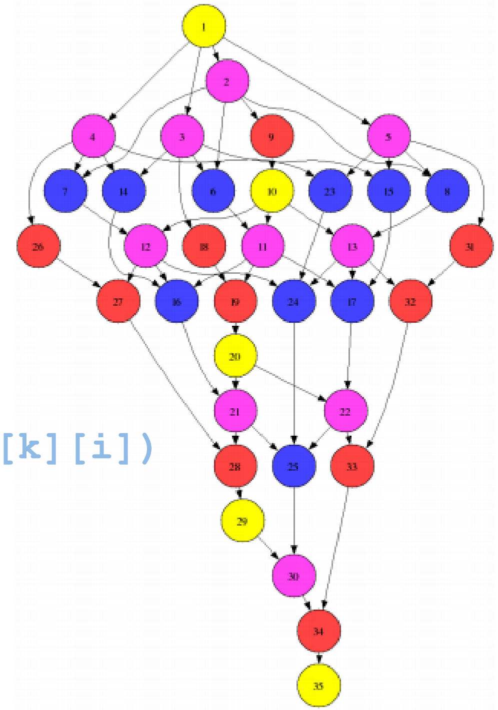
Concurrent Execution w/ Dep.

- **The following allows for even more parallelism, as there now can be two tasks active per thread per i-th iteration.**

```
void process_in_parallel() {
   #pragma omp parallel
   #pragma omp single
   {
      for (int i = 0; i < T; ++i) {
         #pragma omp task firstprivate(i)
         {
            #pragma omp task depend(out: i)
                preprocess_some_data(...);
            #pragma omp task depend(in: i)
               do_something_with_data(...);
            #pragma omp task depend(in: i)
               do_something_independent_with_data(...);
         } // end omp task
      }
   } // end omp single, end omp parallel
}
```

„Real“ Task Dependencies

```
void blocked_cholesky( int NB, float A[NB][NB] ) {
   int i, j, k;
   for (k=0; k<NB; k++) {
     #pragma omp task depend(inout:A[k][k])
        spotrf (A[k][k]) ;
     for (i=k+1; i<NT; i++)
       #pragma omp task depend(in:A[k][k]) depend(inout:A[k][i])
          strsm (A[k][k], A[k][i]);
       // update trailing submatrix
       for (i=k+1; i<NT; i++) {
         for (j=k+1; j<i; j++)
           #pragma omp task depend(in:A[k][i],A[k][j])
                            depend(inout:A[j][i])
              sgemm( A[k][i], A[k][j], A[j][i]);
         #pragma omp task depend(in:A[k][i]) depend(inout:A[i][i])
            ssyrk (A[k][i], A[i][i]);
       }
   }
}
```



* image from BSC

User Defined Reductions

Atomics

User Defined Reductions (UDRs) expand OpenMP's usability

- **Use `declare reduction` directive to define operators**
- **Operators used in reduction clause like predefined ops**

```
#pragma omp declare reduction (reduction-identifier :
typename-list : combiner) [initializer(initializer-expr)]
```

- **`reduction-identifier` gives a name to the operator**
  - → Can be overloaded for different types
  - → Can be redefined in inner scopes
- **`typename-list` is a list of types to which it applies**
- **`combiner` expression specifies how to combine values**
- **`initializer` specifies the operator's identity value**
  - → `initializer-expression` is an expression or a function

A simple UDR example

- **Declare the reduction operator**

```
#pragma omp declare reduction (merge : std::vector<int> :
   omp_out.insert(omp_out.end(), omp_in.begin(), omp_in.end()))
```

- **Use the reduction operator in a `reduction` clause**

```
void schedule (std::vector<int> &v, std::vector<int> &filtered) {
  #pragma omp parallel for reduction (merge : filtered)
  for (std:vector<int>::iterator it = v.begin(); it < v.end();
it++)
    if ( filter(*it) )  filtered.push_back(*it);
}
```

- **Private copies created for a reduction are initialized to the identity that was specified for the operator and type**
  - → Default identity defined if `identity` clause not present
- **Compiler uses `combiner` to combine private copies**
  - → `omp_out` refers to private copy that holds combined value
  - → `omp_in` refers to the other private copy

The atomic construct supports effcient parallel accesses

- **Use `atomic` construct for mutually exclusive access to a single memory location**

```
#pragma omp atomic [read|write|update|capture]
[seq_cst]
  expression-stmt
```

  → `expression-stmt` restricted based on type of atomic

  → `update`, the default behavior, reads and writes the single memory location atomically

  → `read` reads location atomically

  → `write` writes location atomically

  → `capture` updates or writes location and captures its value (before or after update) into a private variable

Questions?