Advanced OpenMP: NUMA, Vectorization & Tasking

Michael Klemm OpenMP AMD

Christian Terboven NHR4CES | RWTH AACHEN UNIVERSITY

Bronis R. de Supinski LAWRENCE LIVERMORE NATIONAL LABORATORY

Xavier Teruel Barcelona Supercomputing Center, Centro Nacional de Supercomputación

IWOMP

Understanding Memory Access

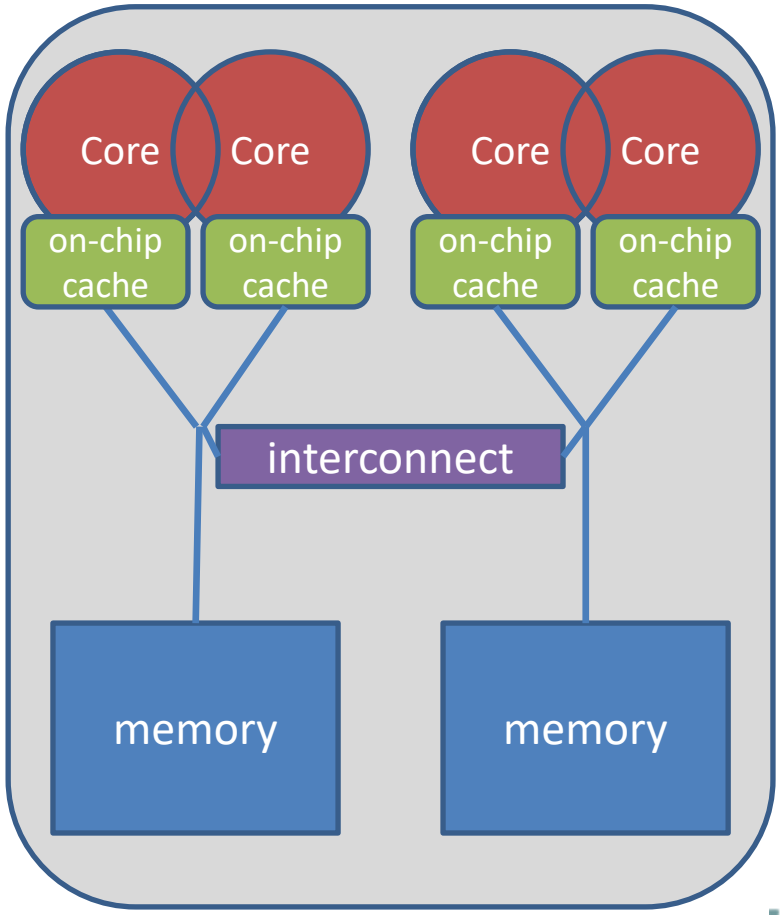
Memory Affinity

Non-uniform Memory

How To Distribute The Data ?

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

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

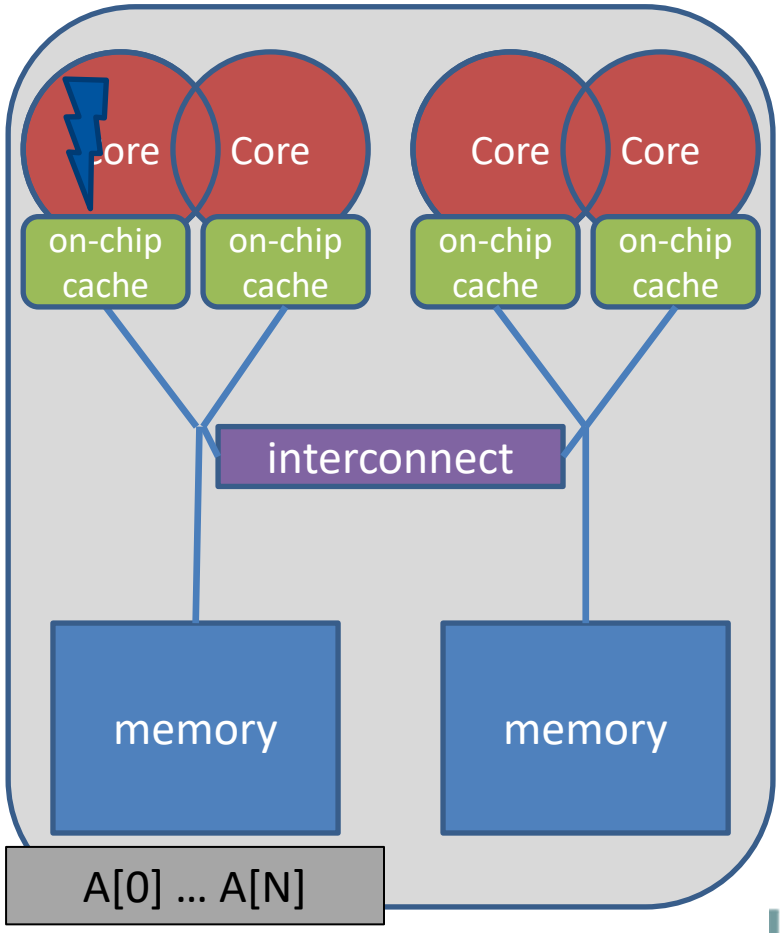


Non-uniform Memory

- Serial code: all array elements are allocated in the memory of the NUMA node closest to the core executing the initializer thread (first touch)

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

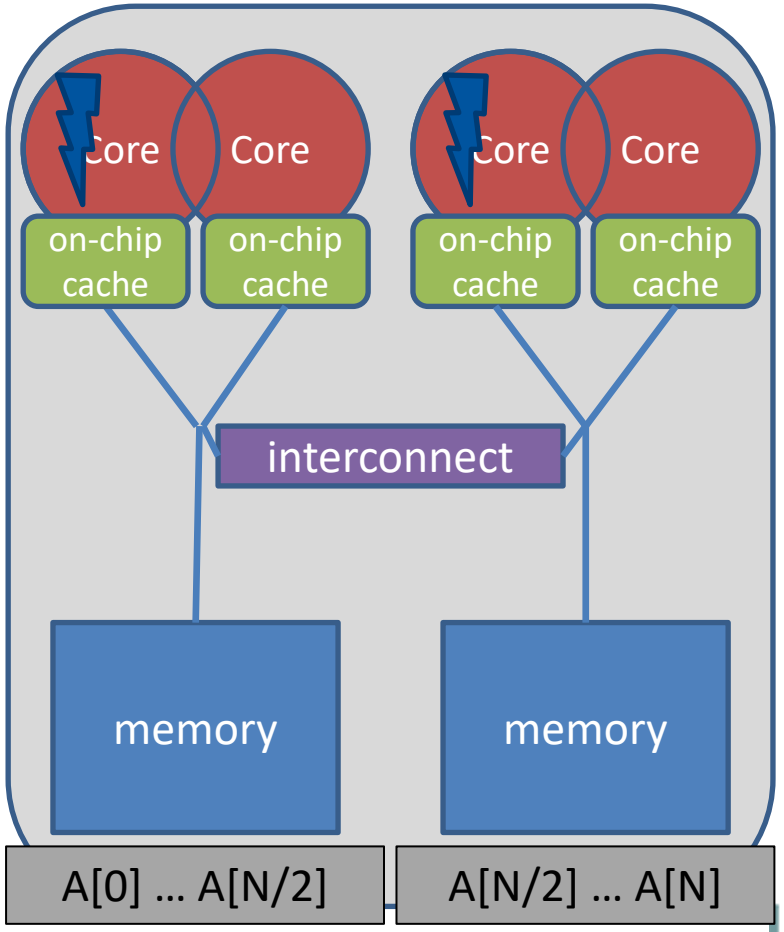
```
for (int i = 0; i < N; i++) {
   A[i] = 0.0;
}
```



First Touch Memory Placement

- First Touch w/ parallel code: all array elements are allocated in the memory of the NUMA node that contains the core that executes the thread that initializes the partition

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

omp_set_num_threads(2);

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

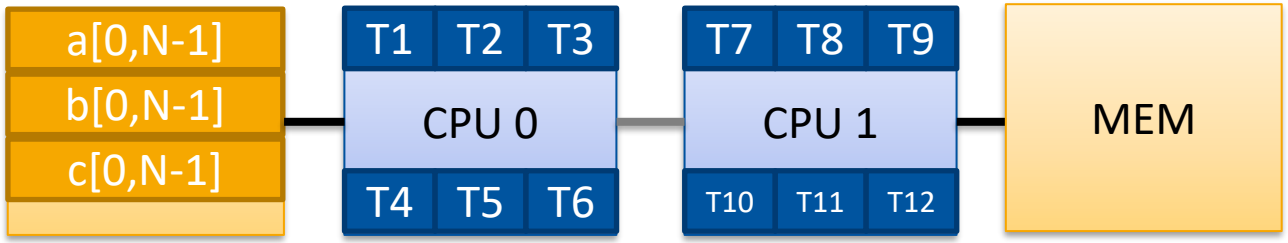


Serial vs. Parallel Initialization

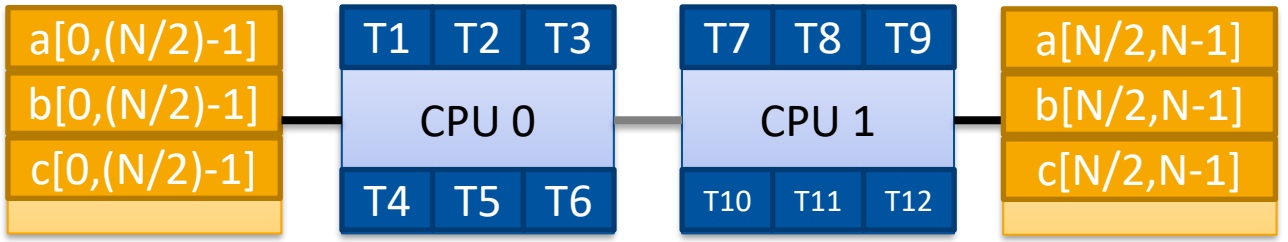
- Stream example with and without parallel initialization.
  - → 2 socket sytem with Xeon X5675 processors, 12 OpenMP threads

	copy	scale	add	triad
ser_init	18.8 GB/s	18.5 GB/s	18.1 GB/s	18.2 GB/s
par_init	41.3 GB/s	39.3 GB/s	40.3 GB/s	40.4 GB/s

ser_init:



par_init:



Thread Binding and Memory Placement

Get Info on the System Topology

- Before you design a strategy for thread binding, you should have a basic understanding of the system topology:
  - → Intel MPI's `cpuinfo` tool
    - → `module switch openmpi intelmpi`
    - → `cpuinfo`
    - → Delivers information about the number of sockets (= packages) and the mapping of processor IDs to CPU cores used by the OS
  - → hwlocs' `hwloc-ls` tool
    - → `hwloc-ls`
    - → Displays a graphical representation of the system topology, separated into NUMA nodes, along with the mapping of processor IDs to CPU cores used by the OS and additional information on caches

Decide for Binding Strategy

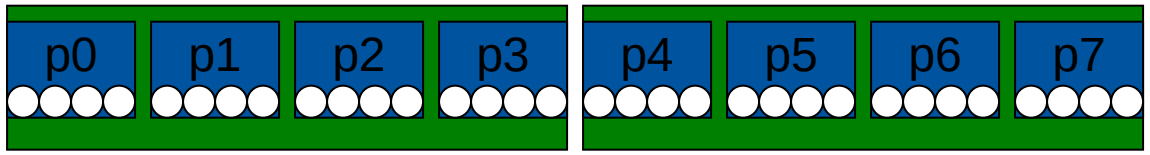
- Selecting the „right“ binding strategy depends not only on the topology, but also on the characteristics of your application.
  - → Putting threads far apart, i.e., on different sockets
    - →May improve the aggregated memory bandwidth available to your application
    - →May improve the combined cache size available to your application
    - →May decrease performance of synchronization constructs
  - → Putting threads close together, i.e., on two adjacent cores that possibly share 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.

Since OpenMP 4.0: Places + Policies

- 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, partition the place list
  - → CLOSE: pack OpenMP threads near primary thread
  - → PRIMARY: collocate OpenMP thread with primary thread
- Goals
  - → user has a way to specify where to execute OpenMP threads for locality between OpenMP threads / less false sharing / memory bandwidth

OMP_PLACES env. variable

- Assume the following machine:



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

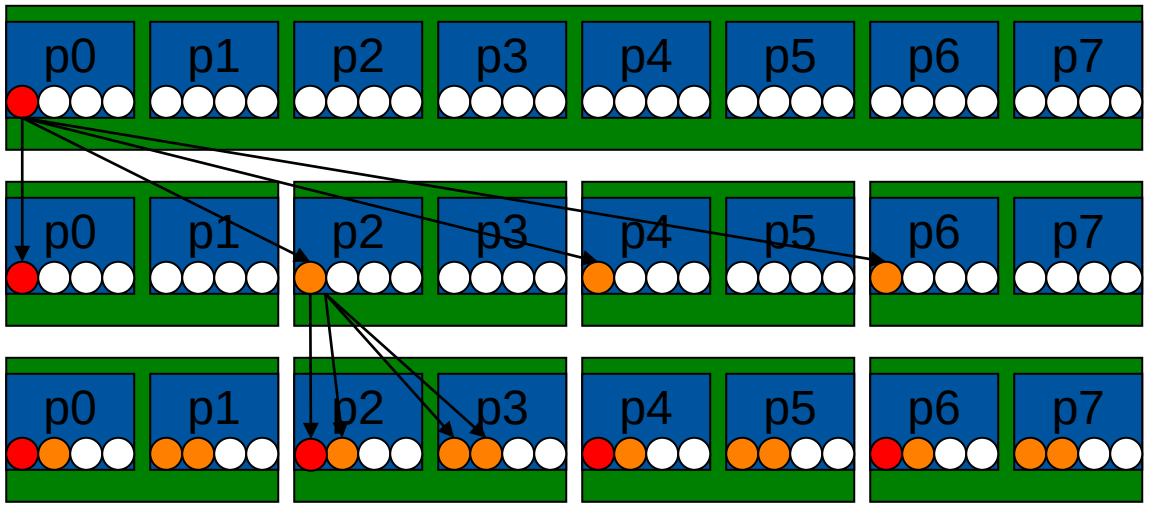
- Abstract names for OMP_PLACES:
  - → threads: Each place corresponds to a single hardware thread.
  - → cores: Each place corresponds to a single core (having one or more hardware threads).
  - → sockets: Each place corresponds to a single socket (consisting of one or more cores).
  - → ll_caches (5.1): Each place corresponds to a set of cores that share the last level cache.
  - → numa_domains (5.1): Each places corresponds to a set of cores for which their closest memory is: the same memory; and at a similar distance from the cores.

OpenMP 4.0: Places + Policies

- 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

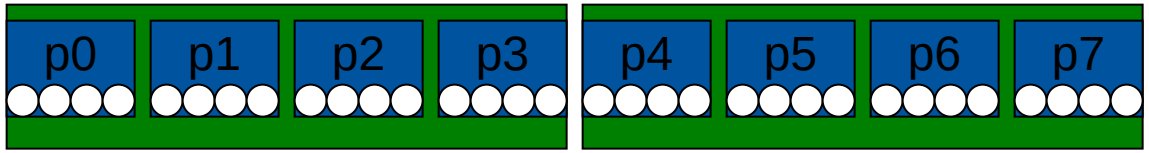
```
OMP_PLACES=(0,1,2,3), (4,5,6,7), ... = (0-4):4:8   = cores
#pragma omp parallel proc_bind(spread) num_threads(4)
#pragma omp parallel proc_bind(close) num_threads(4)
```

- Example
  - → initial
  - → spread 4
  - → close 4



More Examples (1/3)

- Assume the following machine:



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

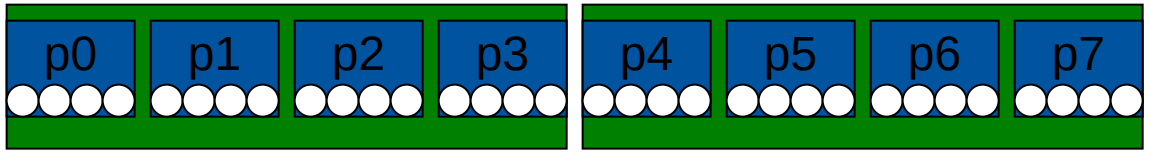
- Parallel Region with two threads, one per socket

→ `OMP_PLACES=sockets`

→ `#pragma omp parallel num_threads(2) proc_bind(spread)`

More Examples (2/3)

- Assume the following machine:



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

- Parallel Region with four threads, one per core, but only on the first socket

→ `OMP_PLACES=cores`

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

More Examples (3/3)

- Spread a nested loop first across two sockets, then among the cores within each socket, only one thread per core
  - → `OMP_PLACES=cores`
  - → `#pragma omp parallel num_threads(2) proc_bind(spread)`
  - → `#pragma omp parallel num_threads(4) proc_bind(close)`

- Places API routines allow to
  - → query information about binding…
  - → query information about the place partition…

Places API: Example

- Simple routine printing the processor ids of the place the calling thread is bound to:

```
void print_binding_info() {
     int my_place = omp_get_place_num();
     int place_num_procs = omp_get_place_num_procs(my_place);

     printf("Place consists of %d processors: ", place_num_procs);

     int *place_processors = malloc(sizeof(int) * place_num_procs);
     omp_get_place_proc_ids(my_place, place_processors)

     for (int i = 0; i < place_num_procs - 1; i++) {
             printf("%d ", place_processors[i]);
     }
     printf("\n");

     free(place_processors);
}
```

OpenMP 5.x way to do this

- Set `OMP_DISPLAY_AFFINITY=TRUE`
  - → Instructs the runtime to display formatted affinity information
  - → Example output for two threads on two physical cores:

```
nesting_level=   1,   thread_num=   0,   thread_affinity=   0,1
nesting_level=   1,   thread_num=   1,   thread_affinity=   2,3
```

  - → Output (
    corresponding routine
  - → Formatted affinity information can be printed with
    `omp_display_affinity(const char* format)`

Affinity format specification

t	omp_get_team_num()
T	omp_get_num_teams()
L	omp_get_level()
n	omp_get_thread_num()
N	omp_get_num_threads()

a	omp_get_ancestor_thread_num() at level-1
H	hostname
P	process identifier
i	native thread identifier
A	thread affinity: list of processors (cores)

- Example:

```
OMP_AFFINITY_FORMAT="Affinity: %0.3L %.8n %.15{A} %.12H"
```

→Possible output:

```
Affinity: 001          0       0-1,16-17       host003
Affinity: 001          1       2-3,18-19       host003
```

Fine-grained control of Memory Affinity

- Explicit NUMA-aware memory allocation:
  - → By carefully touching data by the thread which later uses it
  - → By changing the default memory allocation strategy
    - → Linux: `numactl` command
  - → By explicit migration of memory pages
    - → Linux: `move_pages()`

- Example: using numactl to distribute pages round-robin:
  - → `numactl –interleave=all ./a.out`

Memory Management

Different kinds of memory

- Traditional DDR-based memory
- High-bandwidth memory
- Non-volatile memory
- …

Cascade Lake (Leonide at INRIA)

```
CPU: Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz
Freq Govenor: performance
---------------------
available: 4 nodes (0-3)
node 0 cpus: 0 2 4 6 8 10 12 14 16 18
             20 22 24 26 28 30 32 34 36 38
node 0 size: 191936 MB
node 0 free: 178709 MB
node 1 cpus: 1 3 5 7 9 11 13 15 17 19 21 23
             25 27 29 31 33 35 37 39
node 1 size: 192016 MB
node 1 free: 179268 MB
node 2 cpus:
node 2 size: 759808 MB
node 2 free: 759794 MB
node 3 cpus:
node 3 size: 761856 MB
node 3 free: 761851 MB
node distances:
node   0   1   2   3
  0:  10  21  17  28
  1:  21  10  28  17
  2:  17  28  10  28
  3:  28  17  28  10
```

DRAM + Optane

Memory Management

- Allocator := an OpenMP object that fulfills requests to allocate and deallocate storage for program variables
- OpenMP allocators are of type `omp_allocator_handle_t`
- Default allocator for host
  - →via `OMP_ALLOCATOR` env. var. or corresponding API
- OpenMP 5.0 supports a set of memory allocators

OpenMP allocators

- Selection of a certain kind of memory

Allocator name	Storage selection intent
omp_default_mem_alloc	use default storage
omp_large_cap_mem_alloc	use storage with large capacity
omp_const_mem_alloc	use storage optimized for read-only variables
omp_high_bw_mem_alloc	use storage with high bandwidth
omp_low_lat_mem_alloc	use storage with low latency
omp_cgroup_mem_alloc	use storage close to all threads in the contention group of the thread requesting the allocation
omp_pteam_mem_alloc	use storage that is close to all threads in the same parallel region of the thread requesting the allocation
omp_thread_local_mem_alloc	use storage that is close to the thread requesting the allocation

Using OpenMP allocators

- New clause on all constructs with data sharing clauses:
  - → `allocate( [allocator:] list )`
- Allocation:
  - → `omp_alloc(size_t size, omp_allocator_handle_t allocator)`
- Deallocation:
  - → `omp_free(void *ptr, const omp_allocator_handle_t allocator)`

- `allocate` directive: standalone directive for allocation, or declaration of allocation stmt.

OpenMP allocator traits / 1

- Allocator traits control the behavior of the allocator

sync_hint	contended, uncontended, serialized, private default: contended
alignment	positive integer value that is a power of two default: 1 byte
access	all, cgroup, pteam, thread default: all
pool_size	positive integer value
fallback	default_mem_fb, null_fb, abort_fb, allocator_fb default: default_mem_fb
fb_data	an allocator handle
pinned	true, false default: false
partition	environment, nearest, blocked, interleaved default: environment

OpenMP allocator traits / 2

- `fallback`: describes the behavior if the allocation cannot be fulfilled
  - → `default_mem_fb`: return system's default memory
  - → Other options: null, abort, or use different allocator

- `pinned`: request pinned memory, i.e. for GPUs

OpenMP allocator traits / 3

- `partition`: partitioning of allocated memory of physical storage resources (think of NUMA)
  - → `environment`: use system's default behavior
  - → `nearest`: most closest memory
  - → `blocked`: partitioning into approx. same size with at most one block per storage resource
  - → `interleaved`: partitioning in a round-robin fashion across the storage resources

Using OpenMP allocator traits

- Construction of allocators with traits via
  - → `omp_allocator_handle_t omp_init_allocator(omp_memspace_handle_t memspace, int ntraits, const omp_alloctrait_t traits[]);`
  - → Selection of memory space mandatory
  - → Empty traits set: use defaults

- Allocators have to be destroyed with `*_destroy_*`

- Custom allocator can be made default with `omp_set_default_allocator(omp_allocator_handle_t allocator)`

Memory Management Status

- **LLVM OpenMP runtime internally already uses libmemkind (libnuma, numactl)**
  - → Support for various kinds of memory: DDR, HBW and Persistent Memory (Optane)
  - → Library loaded at initialization (checks for availability)
  - → If requested memory space for allocator is not available → fallback to DDR

- **Memory Management implementation in LLVM still not complete**
  - → Some allocator traits not implemented yet
  - → Some `partition` values not implemented yet (environment, interleaved, nearest, blocked)
  - → Semantics of `omp_high_bw_mem_space and omp_large_cap_mem_space` unclear. Which memory should be used?
    - → Explicitly target HBM → currently implemented in LLVM

- **LLVM has custom implementation of aligned memory allocation**
  - → Allocation covers → {Allocator Information + Requested Size + Buffer based on alignment}

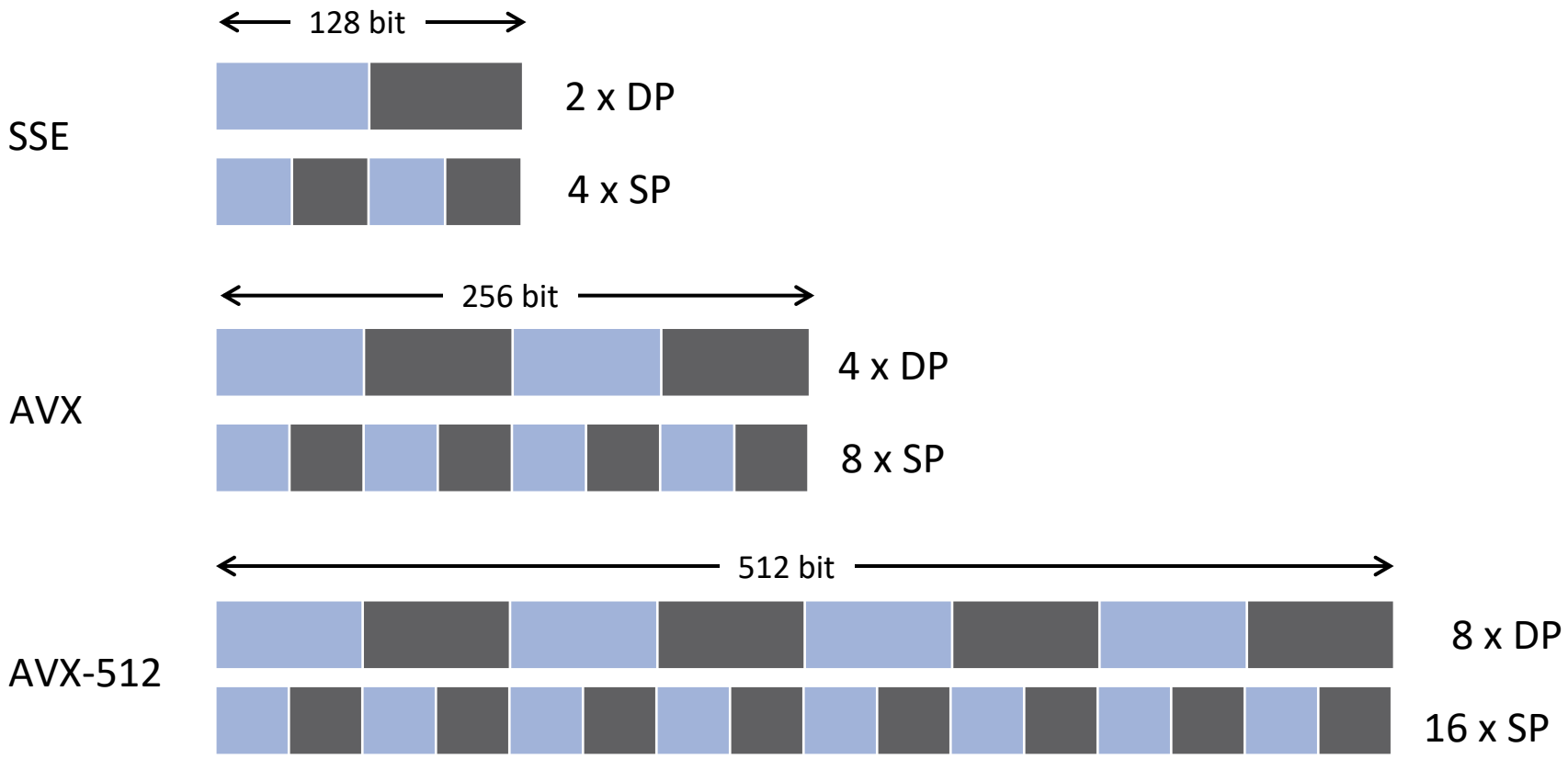
Vectorization

Topics

- Exploiting SIMD parallelism with OpenMP
- Using SIMD directives with loops
- Creating SIMD functions

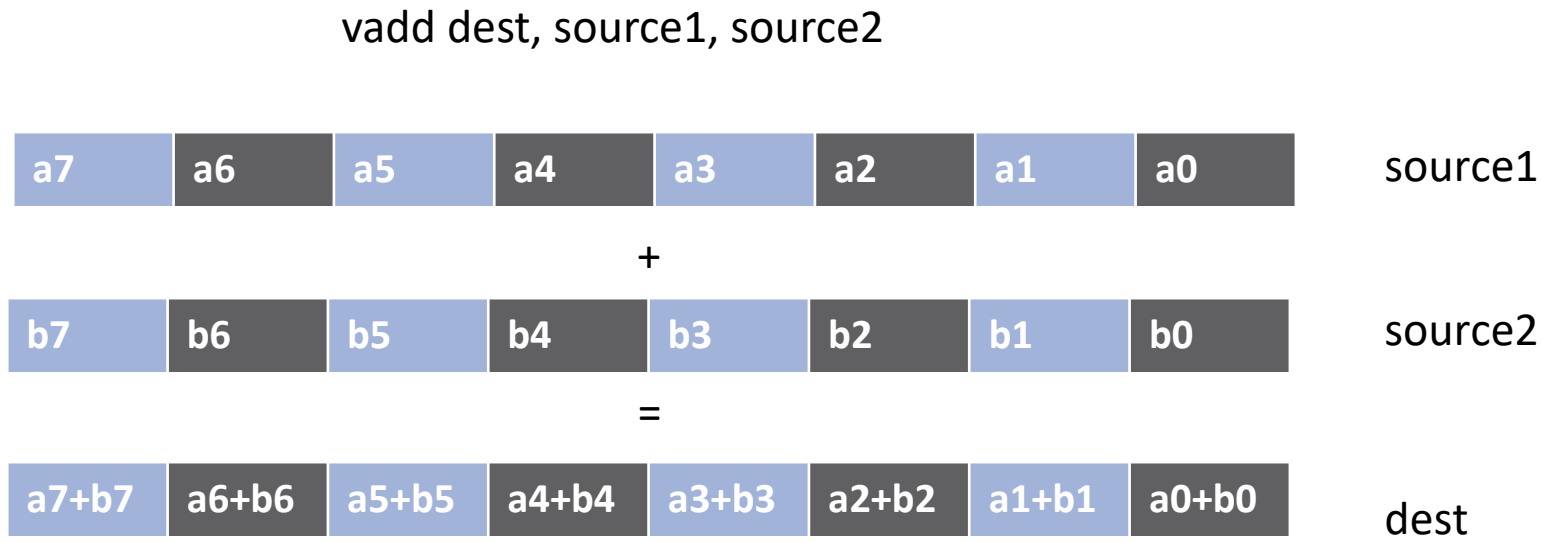
SIMD on x86 Architectures

- Width of SIMD registers has been growing in the past:



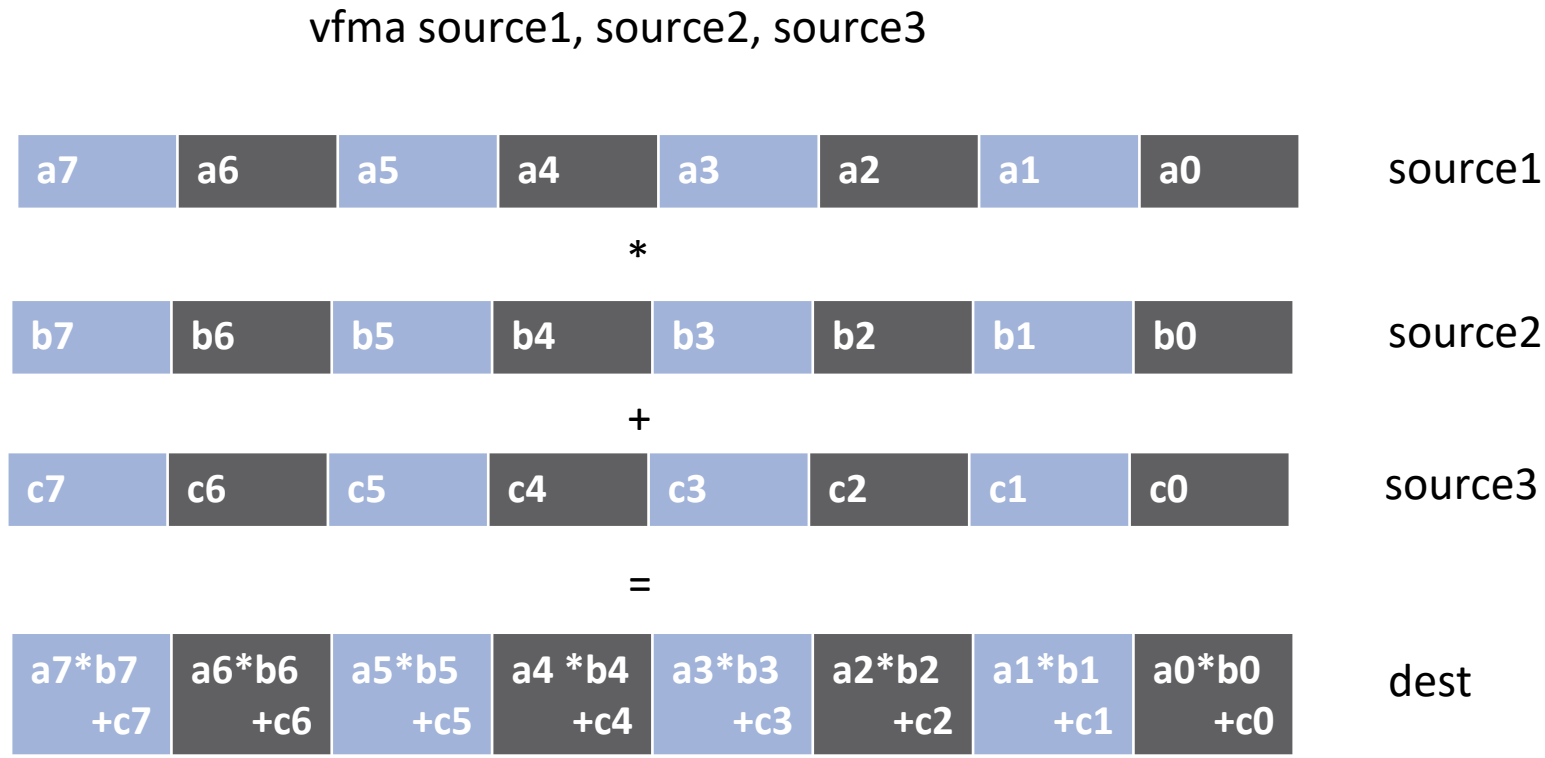
More Powerful SIMD Units

- SIMD instructions become more powerful



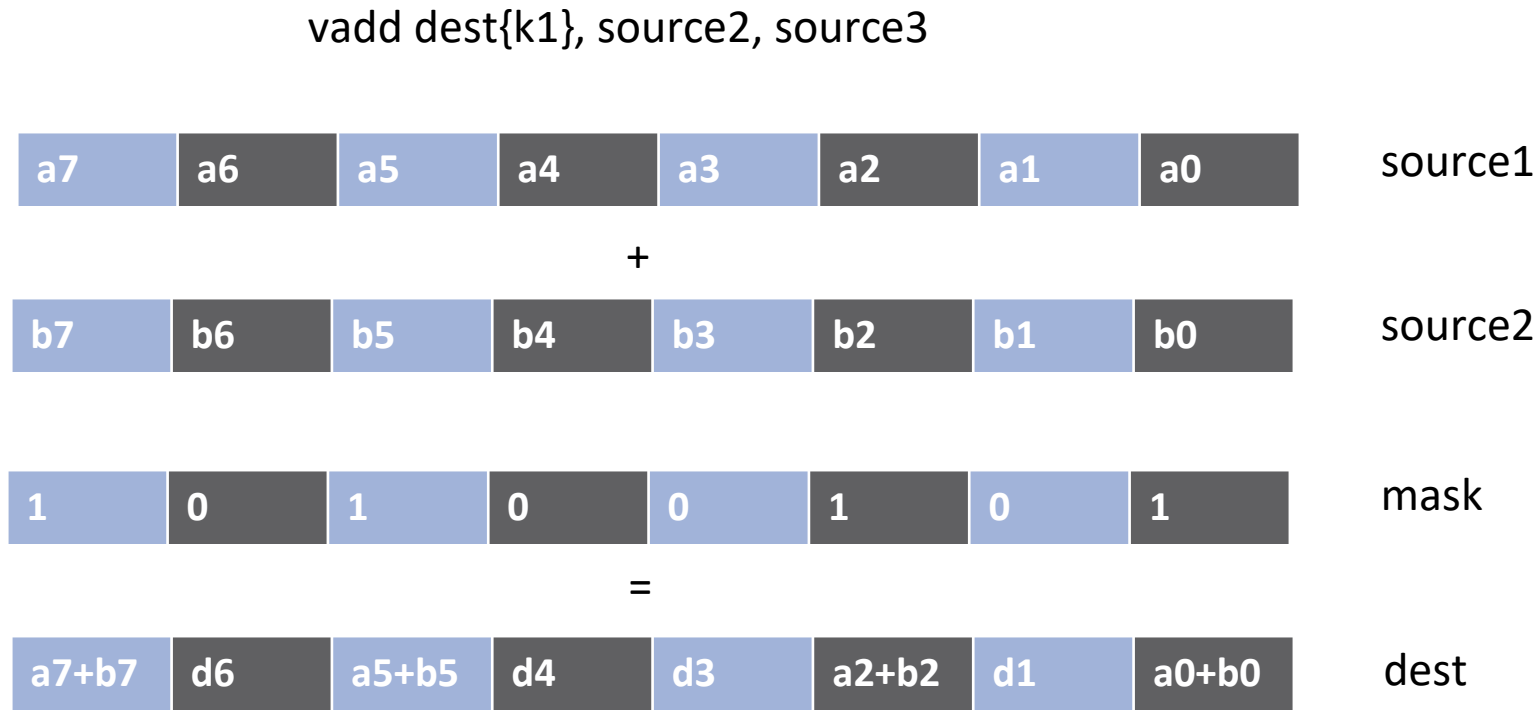
More Powerful SIMD Units

- SIMD instructions become more powerful



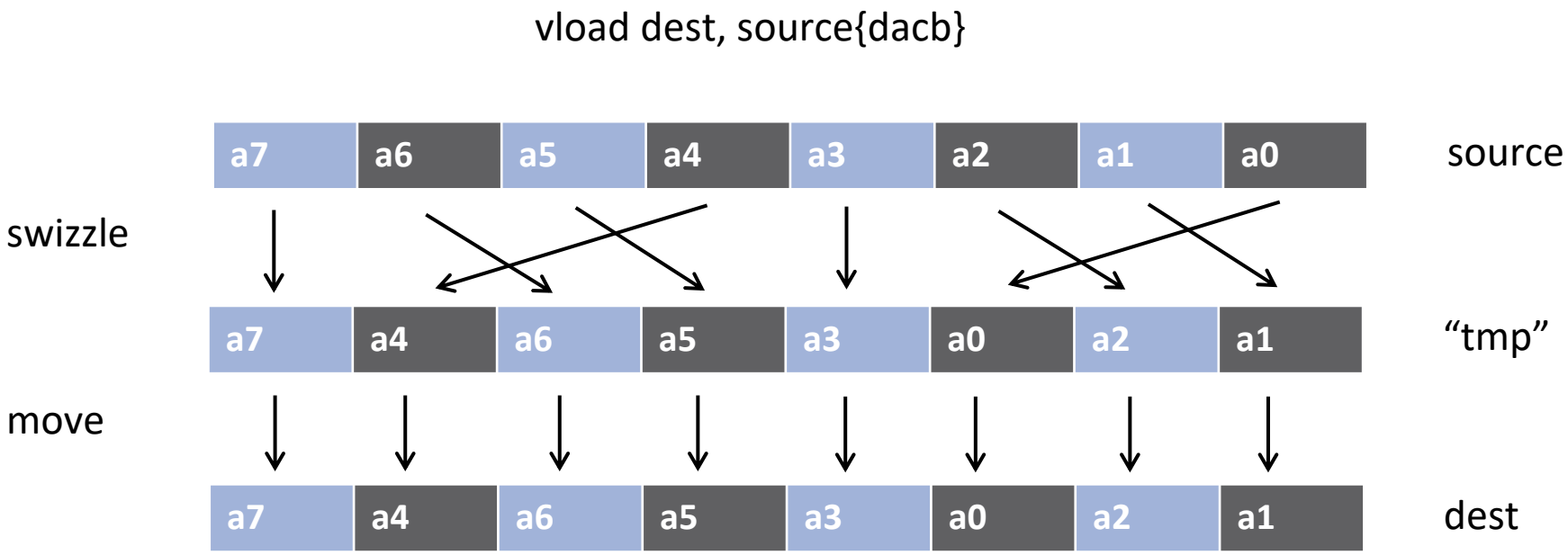
More Powerful SIMD Units

- SIMD instructions become more powerful



More Powerful SIMD Units

- SIMD instructions become more powerful



Auto-vectorization

- Compilers offer auto-vectorization as an optimization pass
  - → Usually, part of the general loop optimization passes
  - → Code analysis detects code properties that inhibit SIMD vectorization ?
  - → Heuristics determine if SIMD execution might be beneficial
  - → If all goes well, the compiler will generate SIMD instructions

Example: clang/LLVM	GCC	Intel Compiler
→ -fvectorize	-ftree-vectorize	-vec (enabled w/ -O2)
→ -Rpass=loop-.*	-ftree-loop-vectorize	-qopt-report=vec
→ -mprefer-vector-width=<*width*>	-fopt-info-vec-all	

Why Auto-vectorizers Fail

- Data dependencies
- Other potential reasons
  - → Alignment
  - → Function calls in loop block
  - → Complex control flow / conditional branches
  - → Loop not “countable”
    - → e.g., upper bound not a runtime constant
  - → Mixed data types
  - → Non-unit stride between elements
  - → Loop body too complex (register pressure)
  - → Vectorization seems inefficient
- Many more … but less likely to occur

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];
    }
}
```

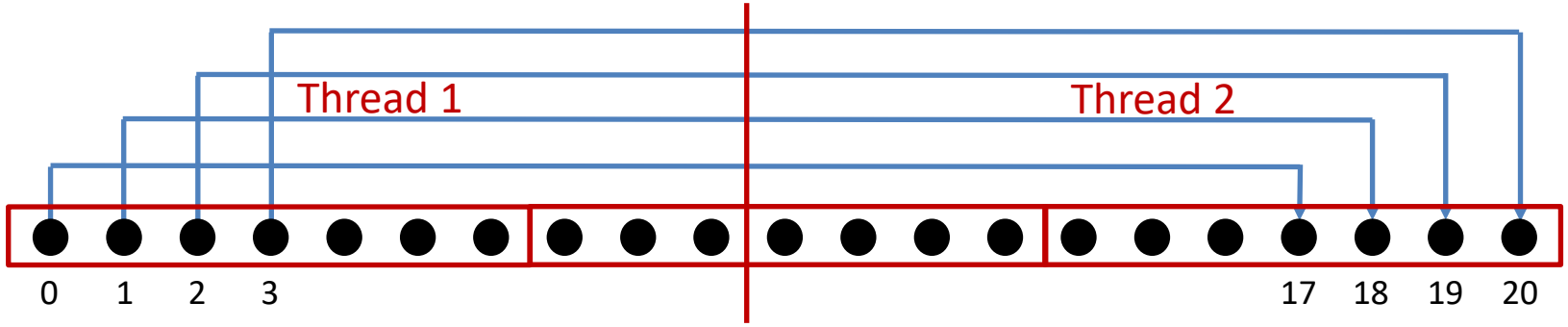
Loop-carried dependency for a[i] and a[i+17]; distance is 17.

- 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];
}   }
```



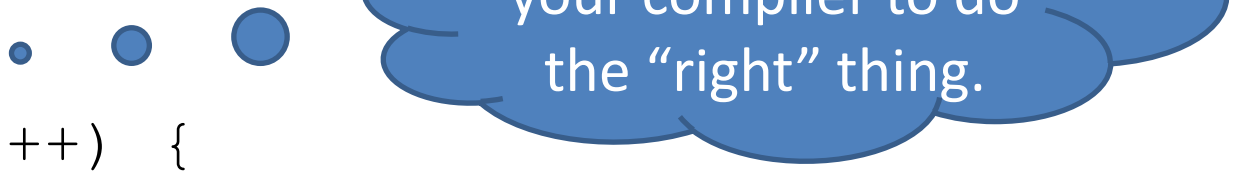
→ Parallelization: no
(except for very specific loop schedules)

→ Vectorization: yes
(iff vector length is shorter than any distance of any dependency)

In a Time Before OpenMP 4.0

- Support required vendor-specific extensions
  - → Programming models (e.g., Intel® Cilk Plus)
  - → Compiler pragmas (e.g., `#pragma vector`)
  - → Low-level constructs (e.g., `_mm_add_pd()`)

```
#pragma omp parallel for
#pragma vector always
#pragma ivdep
for (int i = 0; i < N; i++) {
    a[i] = b[i] + ...;
}
```



SIMD Loop Construct

- Vectorize a loop nest
  - → Cut loop into chunks that fit a SIMD vector register
  - → No parallelization of the loop body

- Syntax (C/C++)

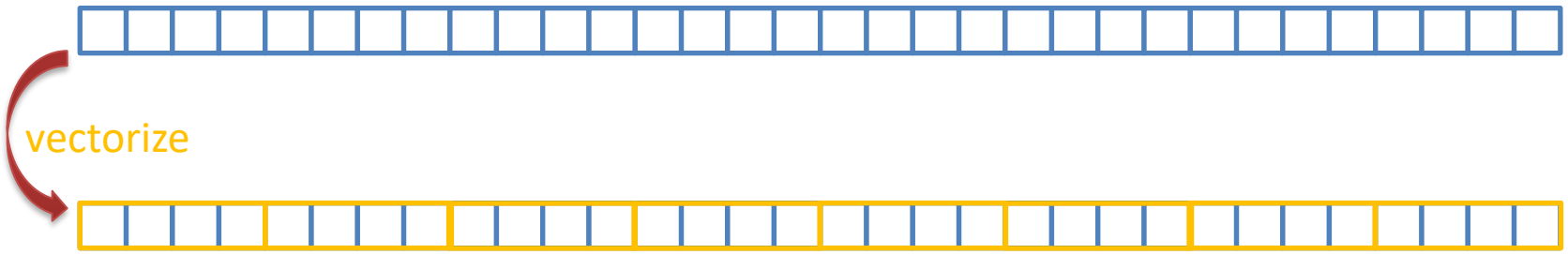
```
#pragma omp simd [clause[[,] clause],…]
for-loops
```

- Syntax (Fortran)

```
!$omp simd [clause[[,] clause],…]
do-loops
[!$omp end simd]
```

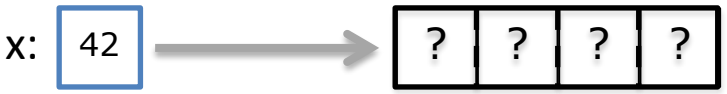
Example

```
float sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp simd reduction(+:sum)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```

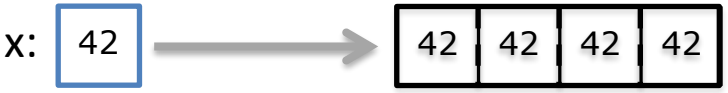


Data Sharing Clauses

- `private(var-list):`
  Uninitialized vectors for variables in *var-list*



- `firstprivate(var-list):`
  Initialized vectors for variables in *var-list*



- `reduction(op:var-list):`
  Create private variables for *var-list* and apply reduction operator *op* at the end of the construct

SIMD Loop Clauses

- `safelen (length)`
  - → Maximum number of iterations that can run concurrently without breaking a dependence
  - → In practice, maximum vector length
- `linear (list[:linear-step])`
  - → The variable's value is in relationship with the iteration number
    - → $x_i = x_{orig} + i * \text{linear-step}$
- `aligned (list[:alignment])`
  - → Specifies that the list items have a given alignment
  - → Default is alignment for the architecture
- `collapse (n)`

SIMD Worksharing Construct

- Parallelize and vectorize a loop nest
  - → Distribute a loop's iteration space across a thread team
  - → Subdivide loop chunks to fit a SIMD vector register

- Syntax (C/C++)

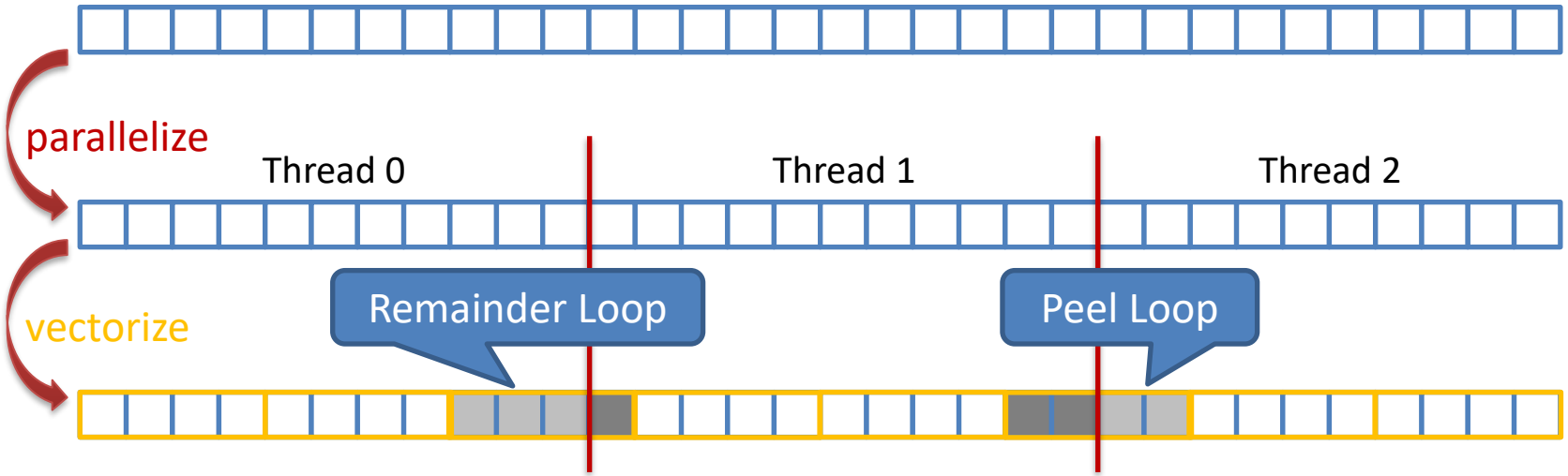
```
#pragma omp for simd [clause[[,] clause],…]
for-loops
```

- Syntax (Fortran)

```
!$omp do simd [clause[[,] clause],…]
do-loops
[!$omp end do simd [nowait]]
```

Example

```
float sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp for simd reduction(+:sum)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```



Be Careful What You Wish For…

```
float sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp for simd reduction(+:sum) \
                     schedule(static, 5)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```

- You should choose chunk sizes that are multiples of the SIMD length
  - → Remainder loops are not triggered
  - → Likely better performance
- In the above example …
  - → and AVX2, the code will only execute the remainder loop!
  - → and SSE, the code will have one iteration in the SIMD loop plus one in the remainder loop!

OpenMP 4.5 Simplifies SIMD Chunks

```
float sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp for simd reduction(+:sum) \
                     schedule(simd: static, 5)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```

- Chooses chunk sizes that are multiples of the SIMD length
  - → First and last chunk may be slightly different to fix alignment and to handle loops that are not exact multiples of SIMD width
  - → Remainder loops are not triggered
  - → Likely better performance

SIMD Function Vectorization

```
float min(float a, float b) {
    return a < b ? a : b;
}

float distsq(float x, float y) {
    return (x - y) * (x - y);
}

void example() {
#pragma omp parallel for simd
    for (i=0; i<N; i++) {
        d[i] = min(distsq(a[i], b[i]), c[i]);
}   }
```

SIMD Function Vectorization

- Declare one or more functions to be compiled for calls from a SIMD-parallel loop

- Syntax (C/C++):

```
#pragma omp declare simd [clause[[,] clause],…]
[#pragma omp declare simd [clause[[,] clause],…]]
[…]
function-definition-or-declaration
```

- Syntax (Fortran):

```
!$omp declare simd (proc-name-list)
```

SIMD Function Vectorization

```
#pragma omp declare simd
float min(float a, float b) {
    return a < b ? a : b;
}

#pragma omp declare simd
float distsq(float x, float y) {
    return (x - y) * (x - y);
}

void example() {
#pragma omp parallel for simd
    for (i=0; i<N; i++) {
        d[i] = min(distsq(a[i], b[i]), c[i]);
}   }
```

```
_ZGVZN16vv_min(%zmm0, %zmm1):
    vminps %zmm1, %zmm0, %zmm0
    ret
```

```
_ZGVZN16vv_distsq(%zmm0, %zmm1):
    vsubps %zmm0, %zmm1, %zmm2
    vmulps %zmm2, %zmm2, %zmm0
    ret
```

```
vmovups (%r14,%r12,4), %zmm0
vmovups (%r13,%r12,4), %zmm1
call _ZGVZN16vv_distsq
vmovups (%rbx,%r12,4), %zmm1
call _ZGVZN16vv_min
```

SIMD Function Vectorization

- `simdlen (`*`length`*`)`
  - → generate function to support a given vector length
- `uniform (`*`argument-list`*`)`
  - → argument has a constant value between the iterations of a given loop
- `inbranch`
  - → function always called from inside an if statement
- `notinbranch`
  - → function never called from inside an if statement
- `linear (`*`argument-list[:linear-step]`*`)`
- `aligned (`*`argument-list[:alignment]`*`)`

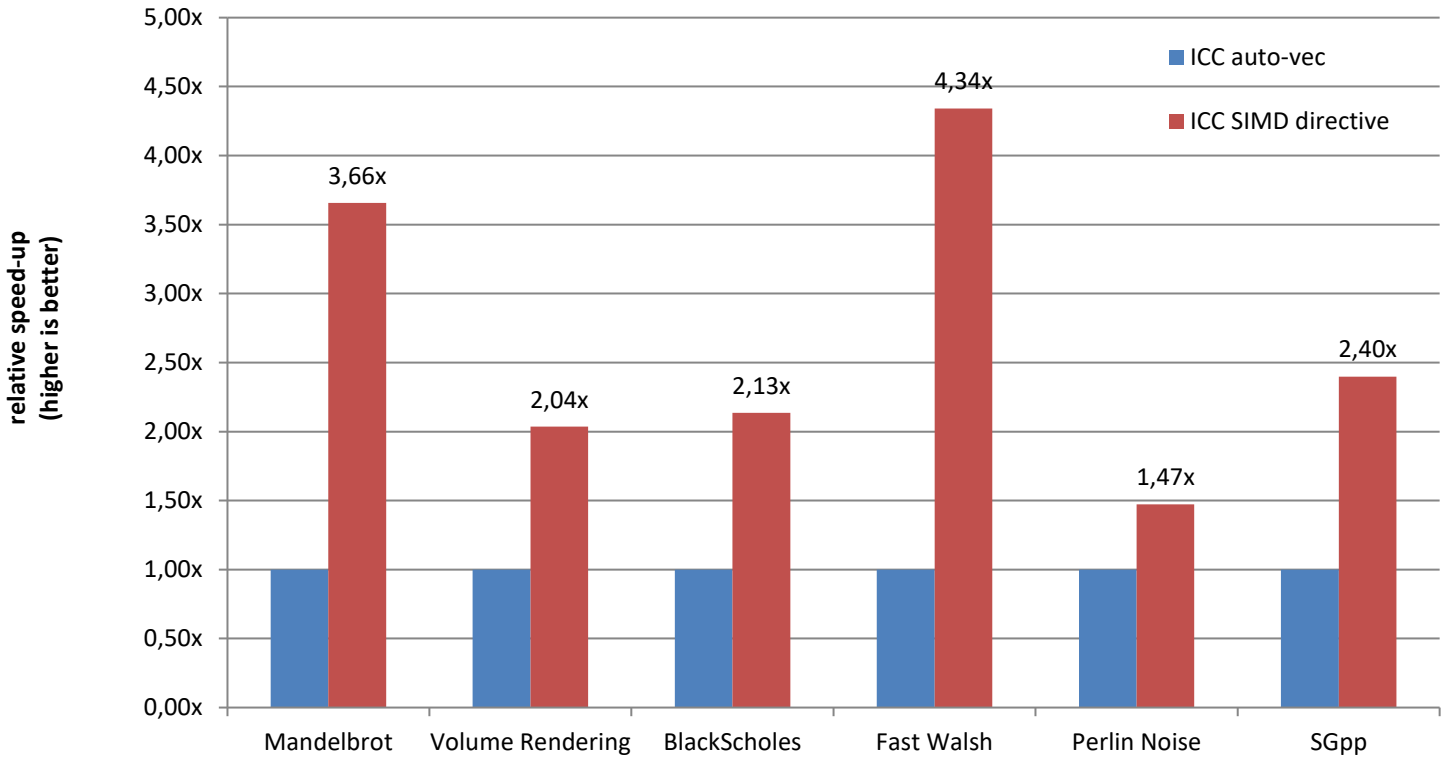
inbranch & notinbranch

```
#pragma omp declare simd inbranch
float do_stuff(float x) {
    /* do something */
    return x * 2.0;
}

void example() {
#pragma omp simd
    for (int i = 0; i < N; i++)
        if (a[i] < 0.0)
            b[i] = do_stuff(a[i]);
}
```

```
vec8 do_stuff_v(vec8 x, mask m) {
    /* do something */
    vmulpd x{m}, 2.0, tmp
    return tmp;
}
```

```
for (int i = 0; i < N; i+=8) {
    vcmp_lt &a[i], 0.0, mask
    b[i] = do_stuff_v(&a[i], mask);
}
```

SIMD Constructs & Performance



M.Klemm, A.Duran, X.Tian, H.Saito, D.Caballero, and X.Martorell. Extending OpenMP with Vector Constructs for Modern Multicore SIMD Architectures. In Proc. of the Intl. Workshop on OpenMP, pages 59-72, Rome, Italy, June 2012. LNCS 7312.

Tasking Part 1:

Introduction, Cutoff, Affinity

Agenda

- Tasking Overview (material compiled by Xavier Teruel @BSC)
- Cutoff clauses and strategies
- Task Affinity

The task construct

- Deferring (or not) a unit of work (executable for any member of the team)

```
#pragma omp task [clause[[,] clause]...]
{structured-block}
```

```
!$omp task [clause[[,] clause]...]
…structured-block…
!$omp end task
```

- Where clause is one of:

→ private(list) → firstprivate(list) → shared(list) → default(shared | none) → in_reduction(r-id: list)	Data Environment
→ allocate([allocator:] list) → detach(event-handler)	Miscellaneous
→ if(scalar-expression) → mergeable → final(scalar-expression)	Cutoff Strategies
→ depend(dep-type: list)	Synchronization
→ untied → priority(priority-value) → affinity(list)	Task Scheduling

Cutoff clauses and strategies

Parallel Brute-force Sudoku

- This parallel algorithm finds all valid solutions

	6						8	11			15	14			16
15	11				16	14				12			6		
13		9	12					3	16	14		15	11	10	
2		16		11		15	10	1							
	15	11	10			16	2	13	8	9	12				
12	13			4	1	5	6	2	3					11	10
5		6	1	12		9		15	11	10	7	16			3
	2				10		11	6		5			13		9
10	7	15	11	16				12	13						6
9						1			2		16	10			11
1		4	6	9	13			7		11		3	16		
16	14			7		10	15	4	6	1				13	8
11	10		15				16	9	12	13			1	5	4
		12		1	4	6		16				11	10		
		5		8	12	13		10			11	2			14
3	16			10			7			6				12	

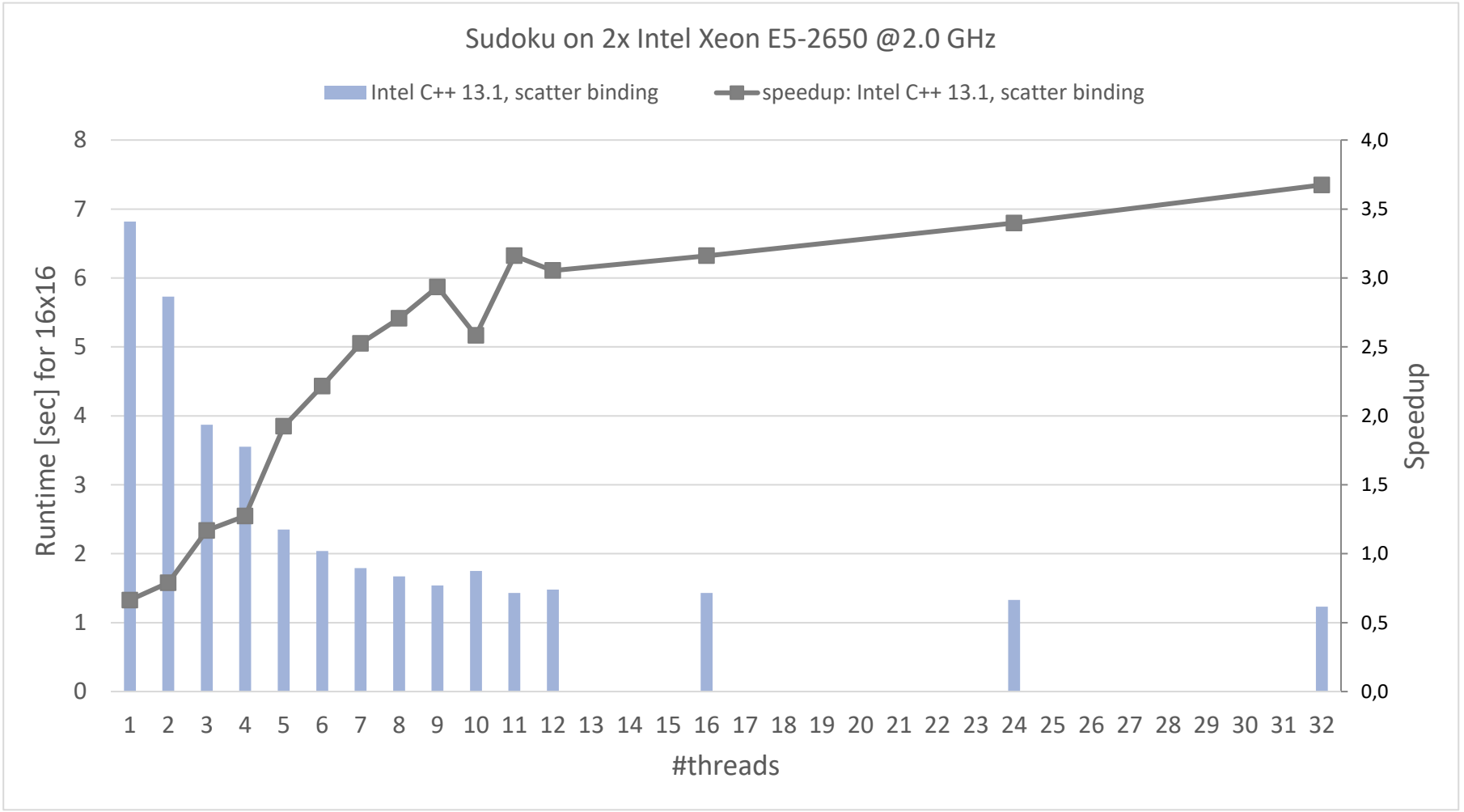
- (1) Search an empty field
- (2) Try all numbers:
  - (2 a) Check Sudoku
    - If invalid: skip
    - If valid: Go to next field
- Wait for completion

first call contained in a
`#pragma omp parallel`
`#pragma omp single`
such that one tasks starts the execution of the algorithm

`#pragma omp task`
needs to work on a new copy of the Sudoku board

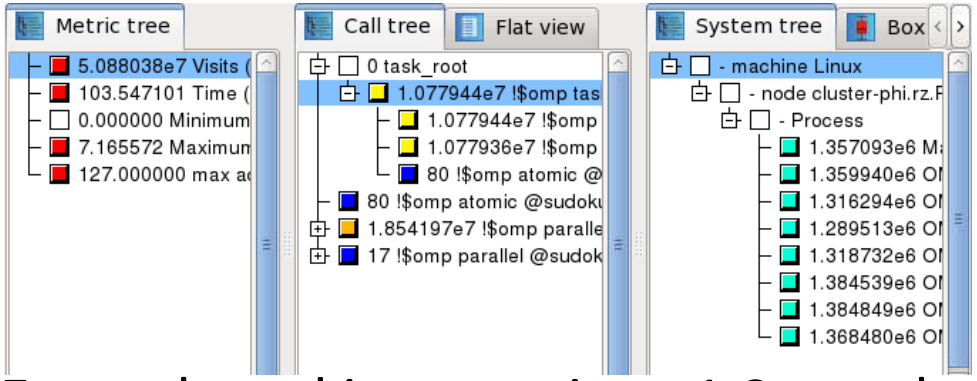
`#pragma omp taskwait`
wait for all child tasks

Performance Evaluation

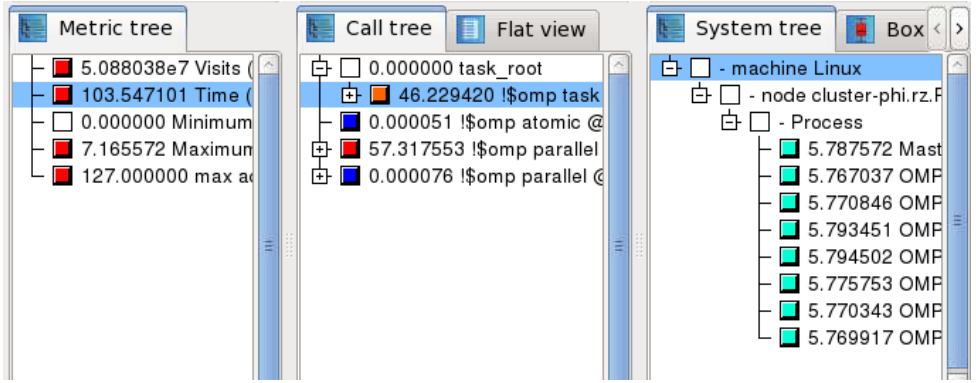


Performance Analysis

Event-based profiling provides a good overview :



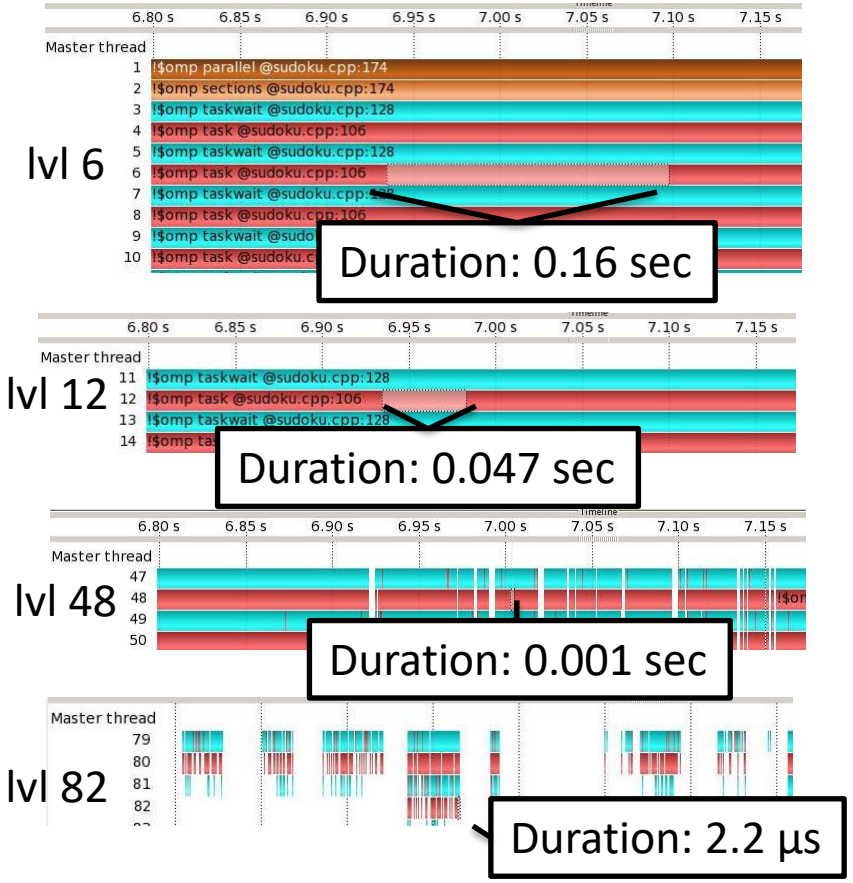
Every thread is executing ~1.3m tasks…



… in ~5.7 seconds.

=> average duration of a task is ~4.4 µs

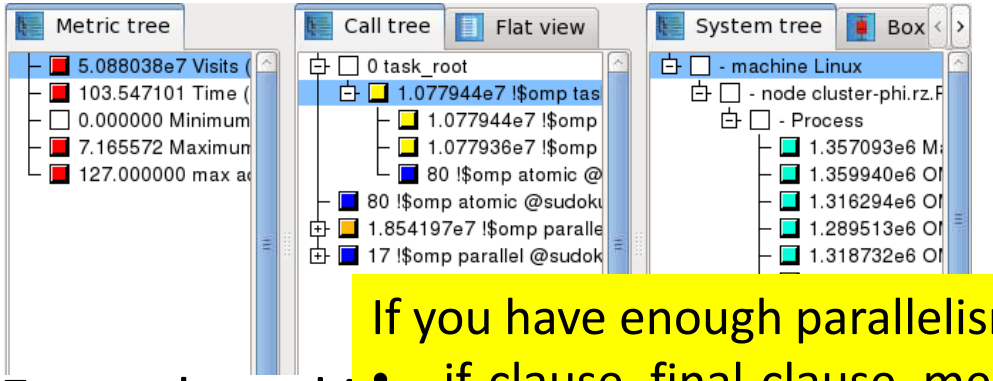
Tracing provides more details:



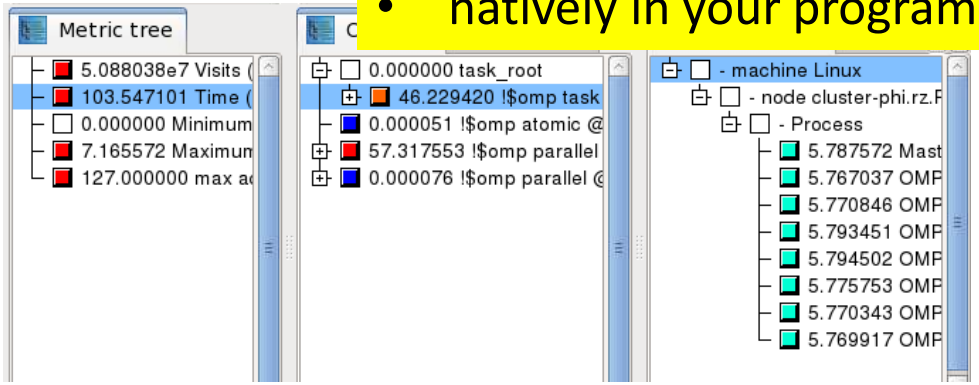
Tasks get much smaller down the call-stack.

Performance Analysis

Event-based profiling provides a good overview :



Every thread i



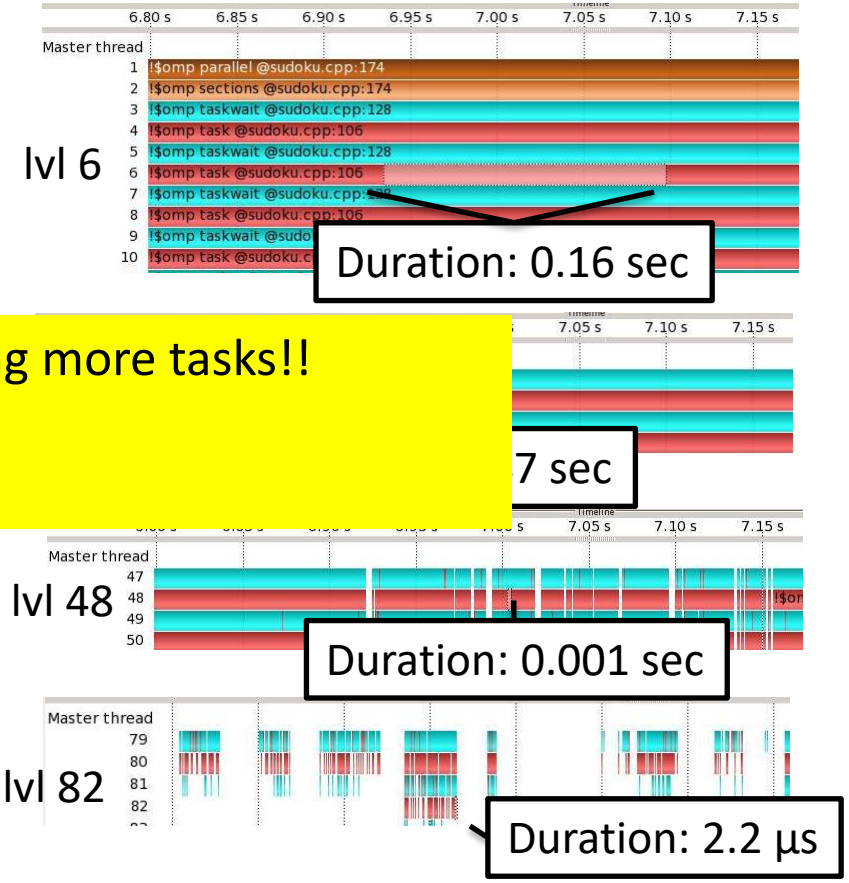
… in ~5.7 seconds.

=> average duration of a task is ~4.4 µs

If you have enough parallelism, stop creating more tasks!!

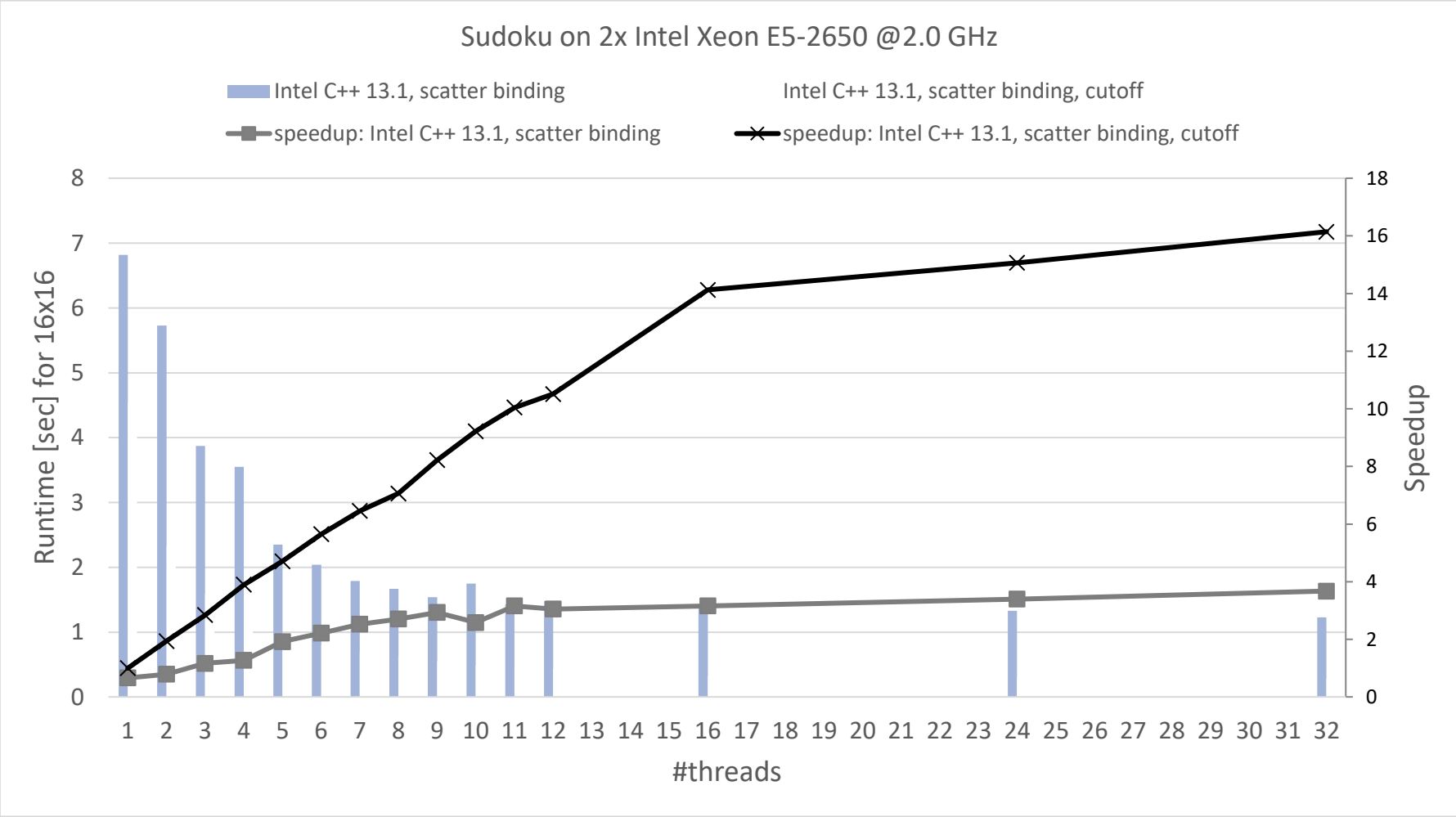
- if-clause, final-clause, mergeable-clause
- natively in your program code

Tracing provides more details:



Tasks get much smaller down the call-stack.

Performance Evaluation (with cutoff)



The `if` clause

- Rule of thumb: the `if(expression)` clause as a "switch off" mechanism
  - → Allows lightweight implementations of task creation and execution but it reduces the parallelism
- If the `expression` of the `if` clause evaluates to `false`
  - → the encountering task is suspended
  - → the new task is executed immediately (task dependences are respected!!)
  - → the encountering task resumes its execution once the new task is completed
  - → This is known as *undeferred task*

```
int foo(int x) {
  printf("entering foo function\n");
  int res = 0;
  #pragma omp task shared(res) if(false)
  {
        res += x;
  }
  printf("leaving foo function\n");
}
```

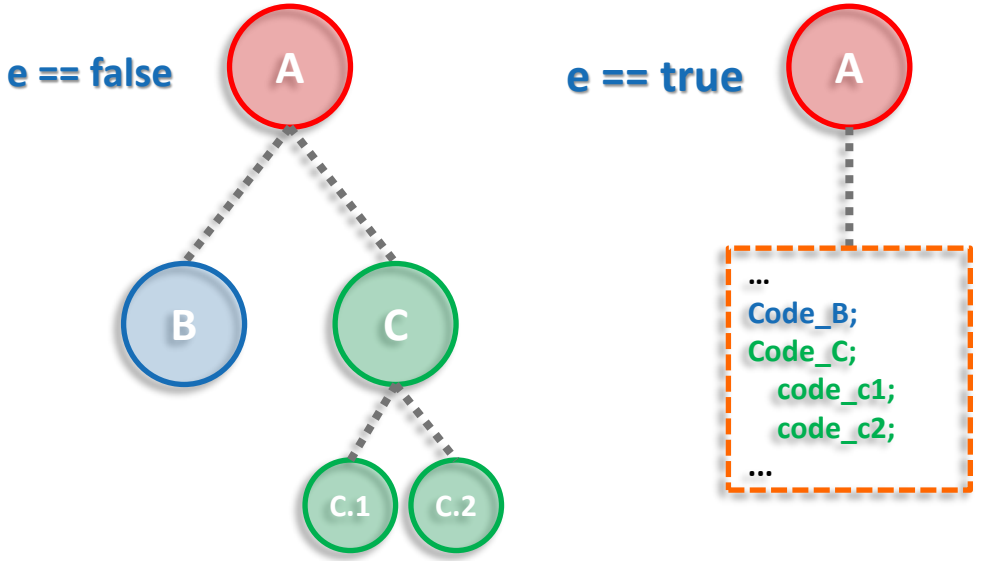
Really useful to debug tasking applications!

- Even if the `expression` is `false`, data-sharing clauses are honored

The `final` clause

- The `final(expression)` clause
  - → Nested tasks / recursive applications
  - → allows to avoid future task creation → reduces overhead but also reduces parallelism
- If the `expression` of the `final` clause evaluates to `true`
  - → The new task is created and executed normally but in its context all tasks will be executed immediately by the same thread (*included tasks*)

```
#pragma omp task final(e)
{
  #pragma omp task
  { … }
  #pragma omp task
  { … #C.1; #C.2 … }
  #pragma omp taskwait
}
```



- Data-sharing clauses are honored too!

The `mergeable` clause

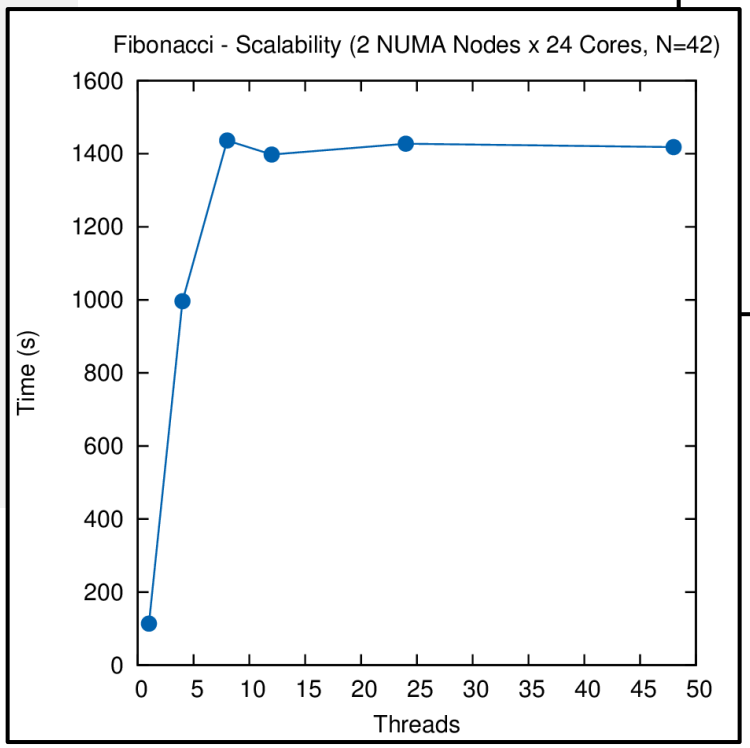
- The `mergeable` clause
  - → Optimization: get rid of “data-sharing clauses are honored”
  - → This optimization can only be applied in *undeferred* or *included tasks*
- A Task that is annotated with the `mergeable` clause is called a *mergeable task*
  - → A task that may be a *merged task* if it is an *undeferred task* or an *included task*
- A *merged task* is:
  - → A task for which the data environment (inclusive of ICVs) may be the same as that of its generating task region
- A good implementation could execute a merged task without adding any OpenMP-related overhead

Unfortunately, there are no OpenMP commercial implementations taking advantatge of `final` neither `mergeable` =(

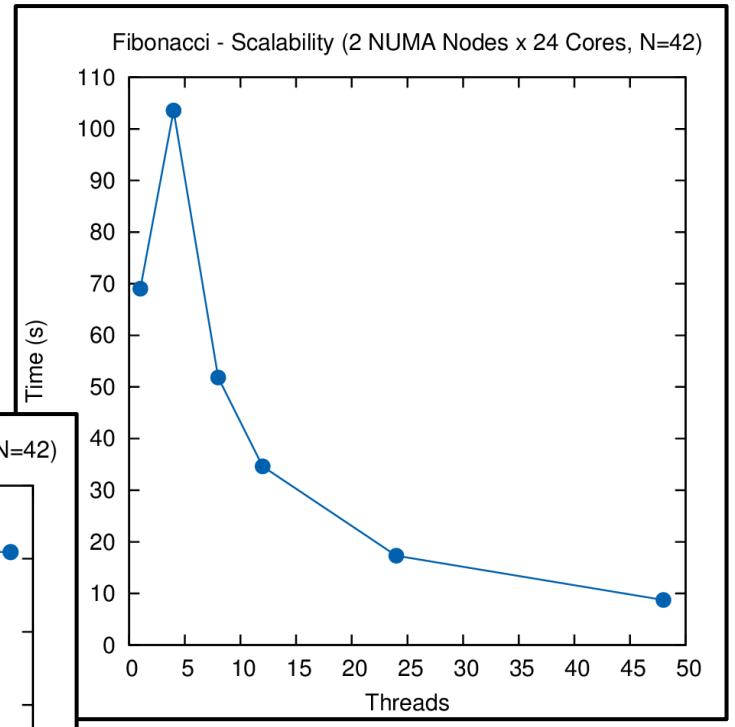
Example: Fibonacci

Fibonacci: without cutoff

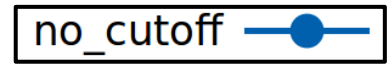
```
int fib(int n) {
  if (n < 2)
    return n;

  int res1, res2;
  #pragma omp task shared(res1)
  res1 = fib(n-1);

  #pragma omp task shared(res2)
  res2 = fib(n-2);

  #pragma omp taskwait

  return res1 + res2;
}
```



gcc 7.2.0

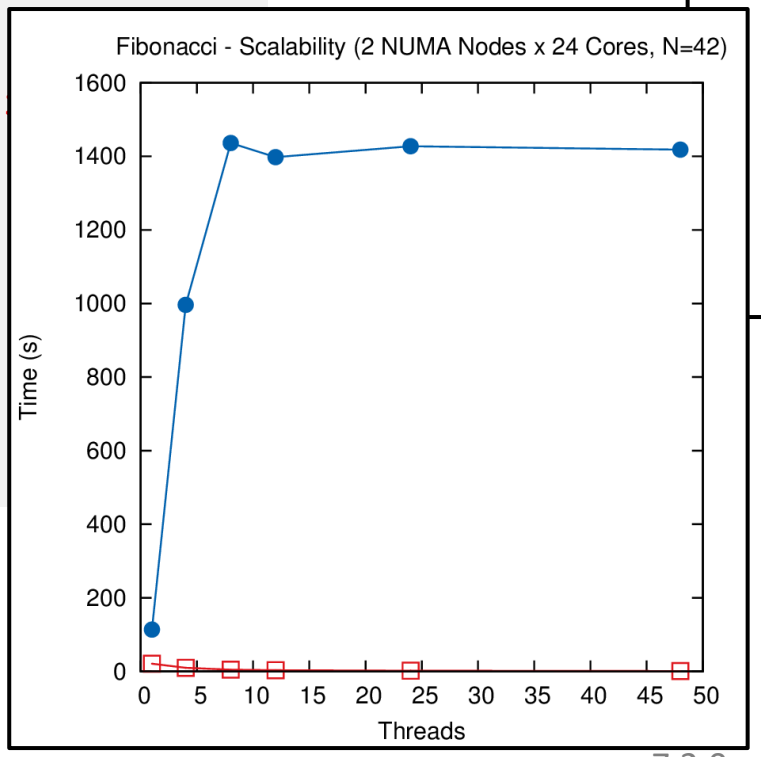


icc 2018.0

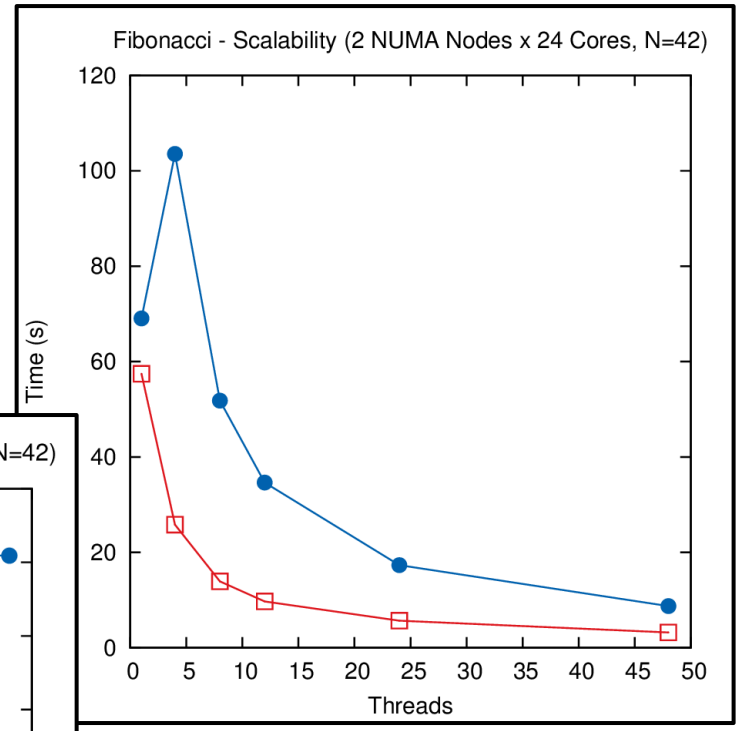


Fibonacci: `if` clause

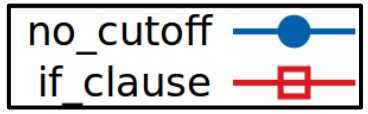
```
int fib(int n) {
  if (n < 2)
    return n;

  int res1, res2;
  #pragma omp task shared(res1) if(n > 30)
  res1 = fib(n-1);

  #pragma omp task shared(res2) if(
  res2 = fib(n-2);

  #pragma omp taskwait

  return res1 + res2;
}
```



gcc 7.2.0

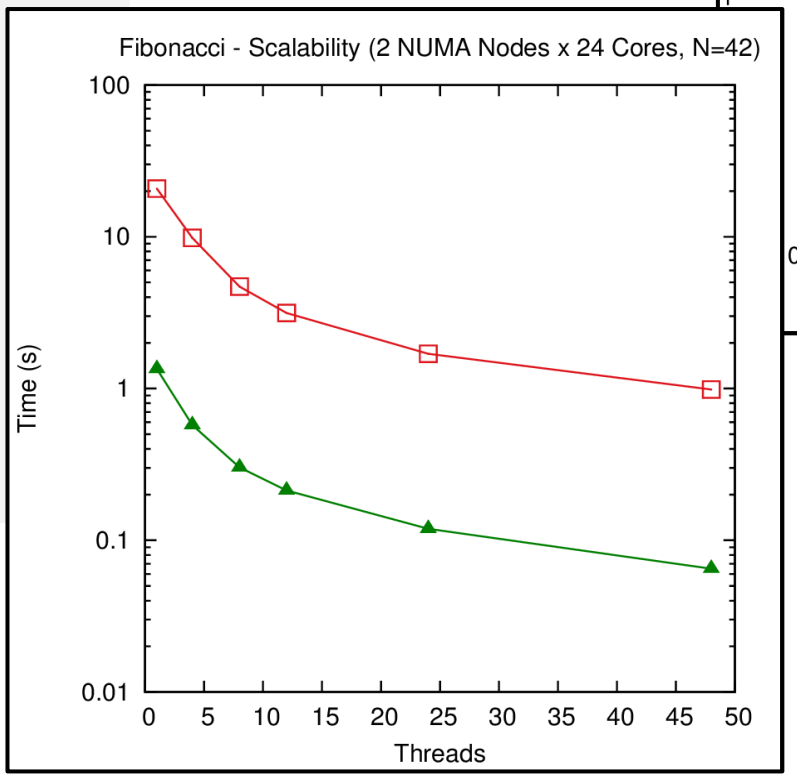


icc 2018.0

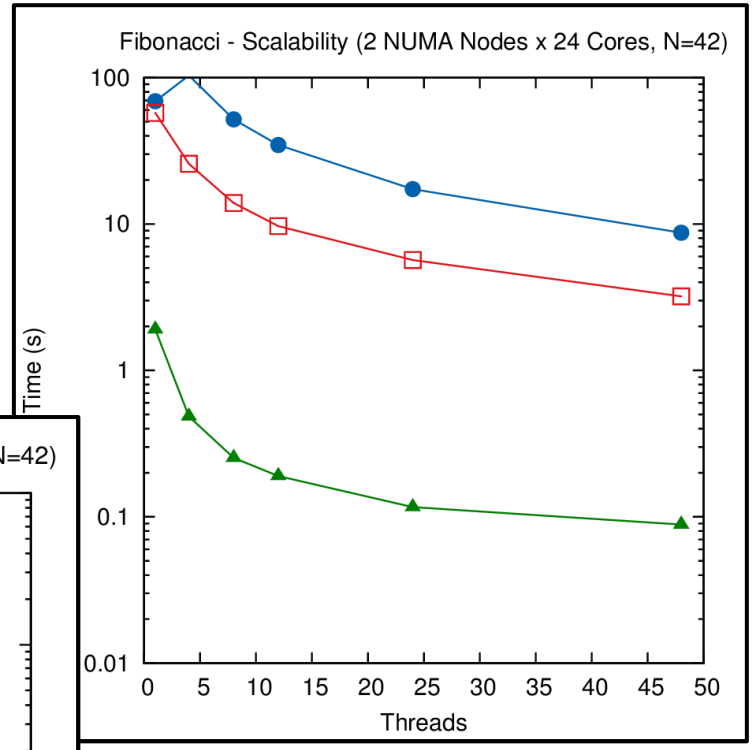


Fibonacci: manual optimization

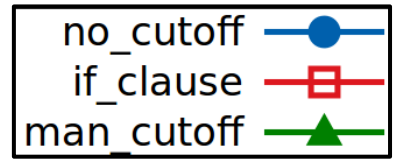
```
int fib(int n) {
  if (n < 30)
    return fib_serial(n);

  int res1, res2;
  #pragma omp task shared(res1)
  res1 = fib(n-1);

  #pragma omp task shared(res2)
  res2 = fib(n-2);

  #pragma omp taskwait

  return res1 + res2;
}
```



gcc 7.2.0



icc 2018.0



Task Affinity

Motivation

- Techniques for process binding & thread pinning available
  - → OpenMP thread level: `OMP_PLACES & OMP_PROC_BIND`
  - → OS functionality: `taskset -c`

<u>OpenMP Tasking:</u>

- In general: Tasks may be executed by any thread in the team
  - → Missing task-to-data affinity may have detrimental effect on performance

<u>OpenMP 5.0:</u>

- `affinity` clause to express affinity to data

`affinity` clause

- New clause: `#pragma omp task affinity (list)`
  - → Hint to the runtime to execute task closely to physical data location
  - → Clear separation between dependencies and affinity
- Expectations:
  - → Improve data locality / reduce remote memory accesses
  - → Decrease runtime variability
- Still expect task stealing
  - → In particular, if a thread is under-utilized

Code Example

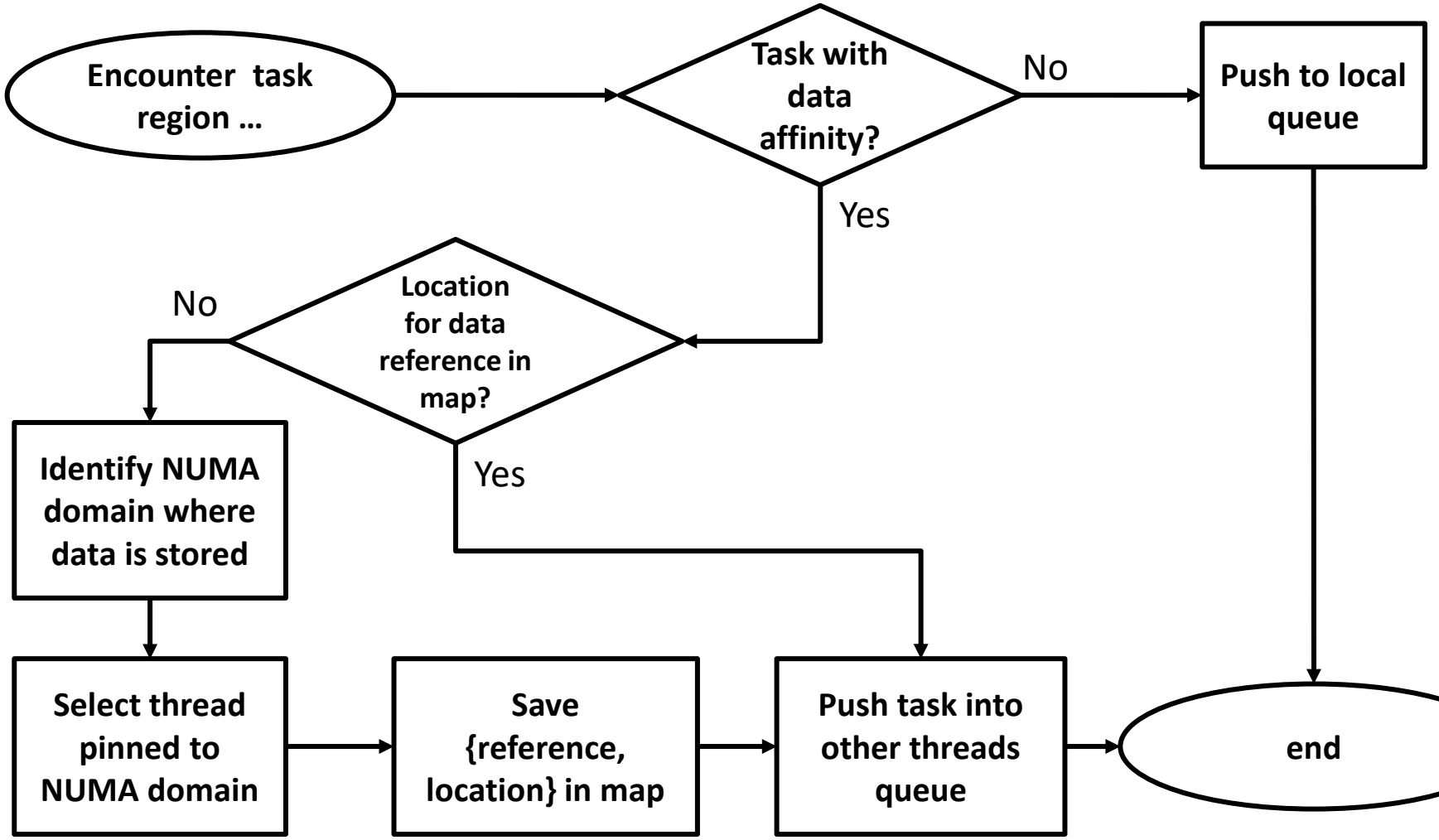
- Excerpt from task-parallel STREAM

```
1   #pragma omp task \
2       shared(a, b, c, scalar) \
3       firstprivate(tmp_idx_start, tmp_idx_end) \
4       affinity( a[tmp_idx_start] )
5   {
6      int i;
7      for(i = tmp_idx_start; i <= tmp_idx_end; i++)
8           a[i] = b[i] + scalar * c[i];
9   }
```

→Loops have been blocked manually (see `tmp_idx_start/end`)

→Assumption: initialization and computation have same blocking and same affinity

Selected LLVM implementation details

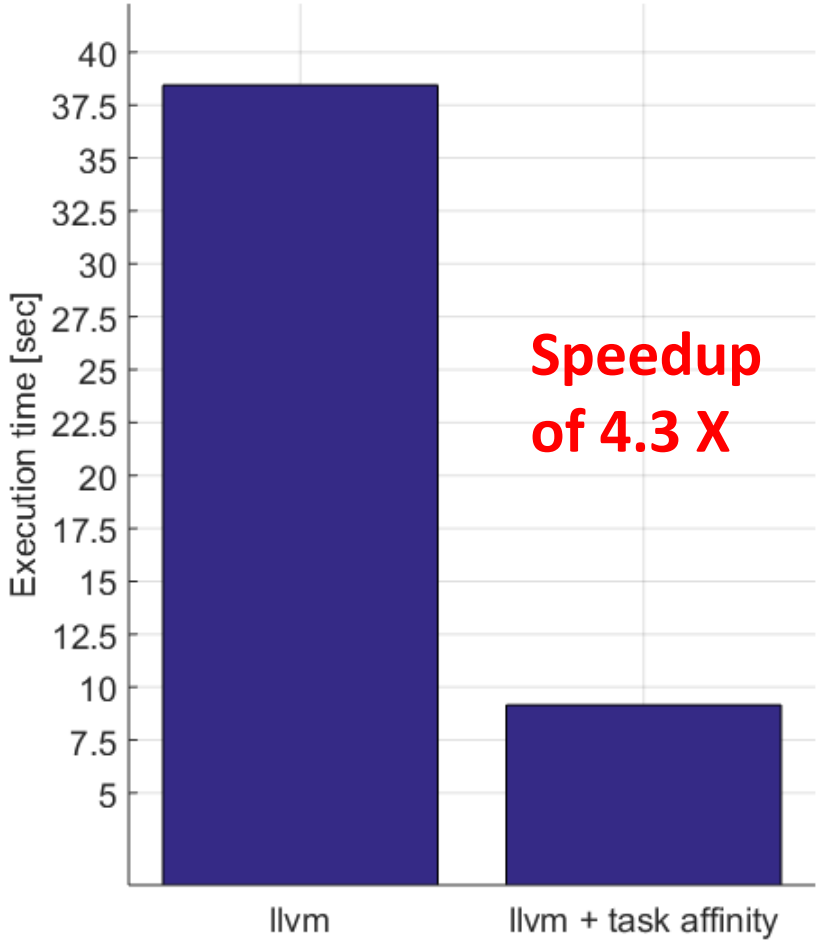


A map is introduced to store location information of data that was previously used

Jannis Klinkenberg, Philipp Samfass, Christian Terboven, Alejandro Duran, Michael Klemm, Xavier Teruel, Sergi Mateo, Stephen L. Olivier, and Matthias S. Müller. **Assessing Task-to-Data Affinity in the LLVM OpenMP Runtime**. Proceedings of the 14th International Workshop on OpenMP, IWOMP 2018. September 26-28, 2018, Barcelona, Spain.

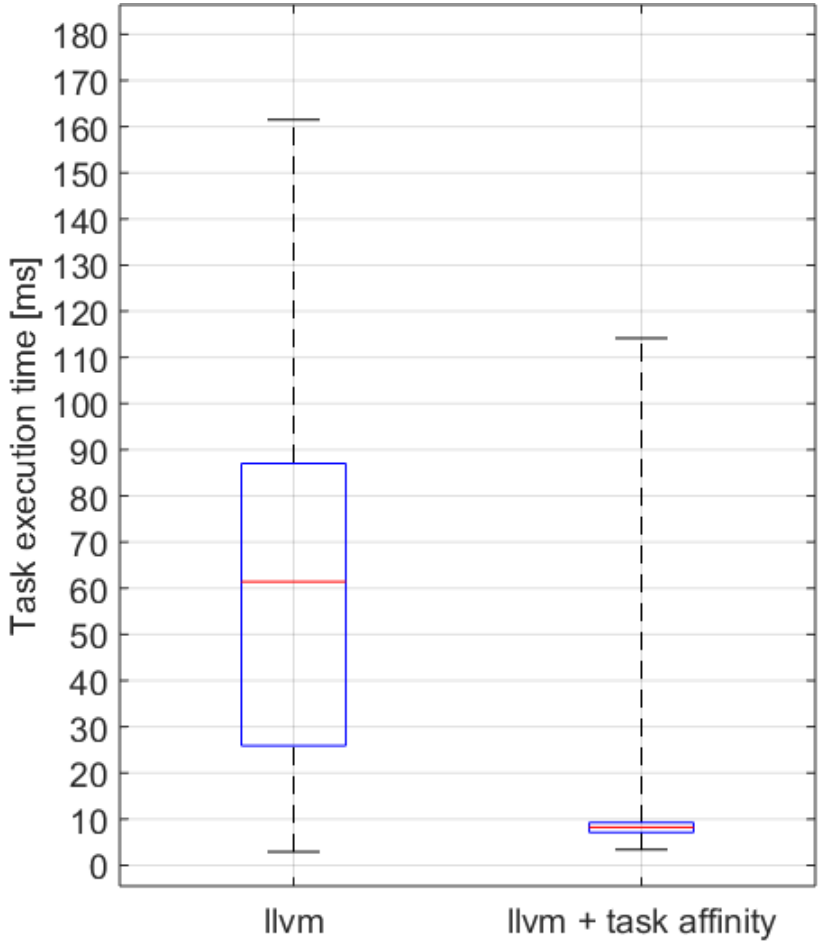
Evaluation

Program runtime
Median of 10 runs



Speedup of 4.3 X

Distribution of single task execution times



LIKWID: reduction of remote data volume from 69% to 13%

Summary

- Requirement for this feature: thread affinity enabled

- The `affinity` clause helps, if
  - → tasks access data heavily
  - → single task creator scenario, or task not created with data affinity
  - → high load imbalance among the tasks

- Different from thread binding: task stealing is absolutely allowed

Tasking Part 2:
Task Dependences

Motivation

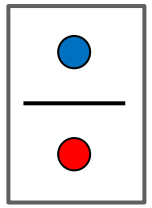
- Task dependences as a way to define task-execution constraints

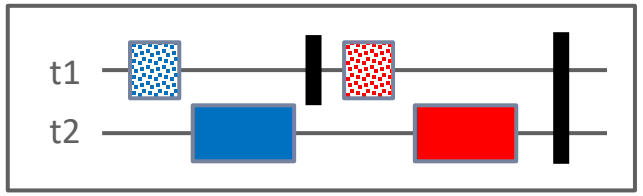
OpenMP 3.1

```
int x = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task
    std::cout << x << std::endl;

    #pragma omp taskwait

  #pragma omp task
    x++;
}
```

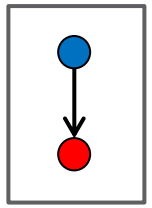
OpenMP 4.0

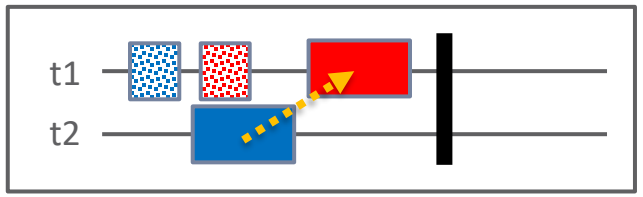
```
int x = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(in: x)
    std::cout << x << std::endl;



  #pragma omp task depend(inout: x)
    x++;
}
```



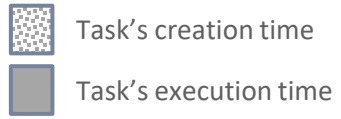


OpenMP 3.1





OpenMP 4.0



Motivation

- Task dependences as a way to define task-execution constraints

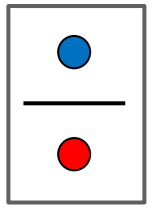
OpenMP 3.1

```
int x = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task
  std::cout << x << std

  #pragma omp taskwait

  #pragma omp task
  x++;
}
```

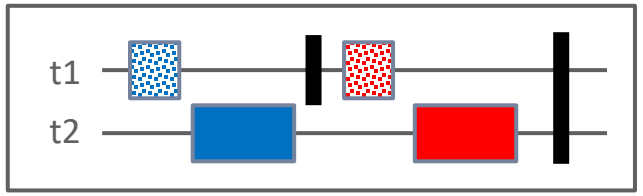
OpenMP 4.0

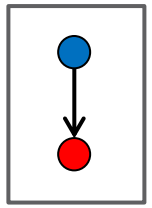
```
int x = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(in: x)
                                d::endl;

                                end(inout: x)
  x++;
}
```

Task dependences can help us to remove "strong" synchronizations, increasing the look ahead and, frequently, the parallelism!!!!

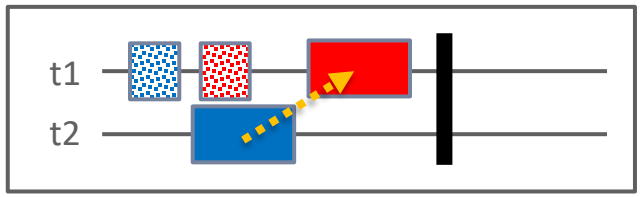


OpenMP 3.1





OpenMP 4.0

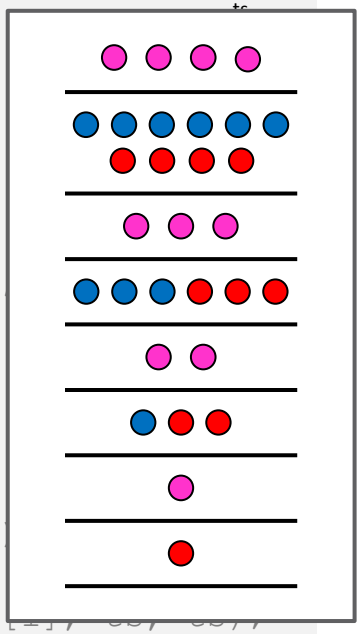


Task's creation time

Task's execution time

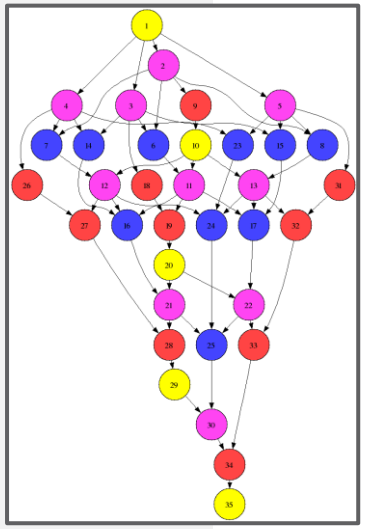
Motivation: Cholesky factorization

```
void cholesky(int ts, int nt, double* a[nt][nt]) {
  for (int k = 0; k < nt; k++) {
    // Diagonal Block factorization
    potrf(a[k][k], ts, ts);

    // Triangular systems
    for (int i = k + 1; i < nt; i++)
      #pragma omp task
      trsm(a[k][k], a[k][i], ts, ts);
    }
    #pragma omp taskwait

    // Update trailing matrix
    for (int i = k + 1; i < nt; i++)
      for (int j = k + 1; j < i; j++
        #pragma omp task
        dgemm(a[k][i], a[k][j], a[j][i],
        }
        #pragma omp task
      syrk(a[k][i], a[i][i], ts, ts);
    }
    #pragma omp taskwait
  }
}
```



OpenMP 3.1

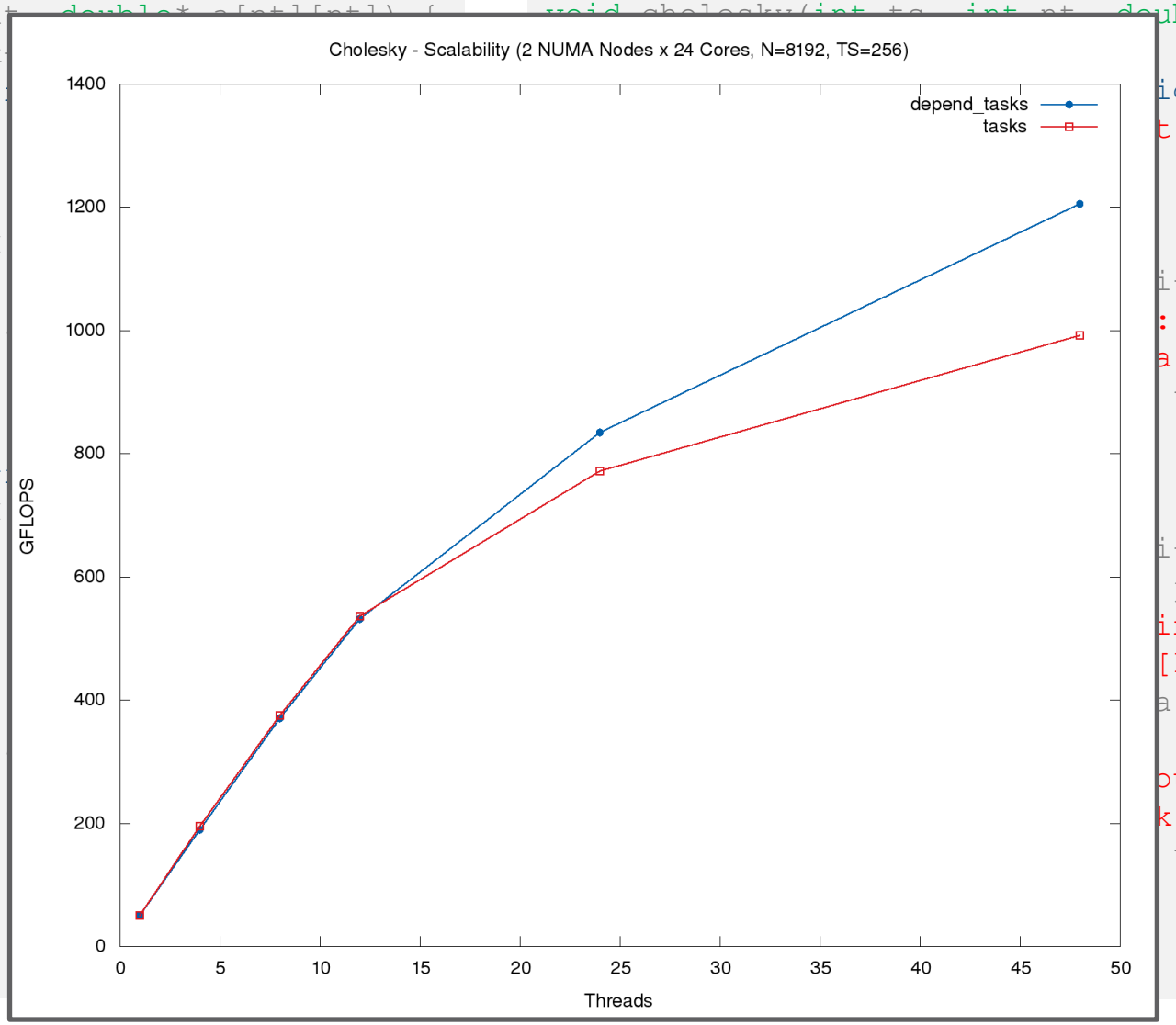
```
void cholesky(int ts, int nt, double* a[nt][nt]) {
  for (int k = 0; k < nt; k++) {
    // Diagonal Block factorization
    #pragma omp task depend(inout: a[k][k])
    potrf(a[k][k], ts, ts);

    // Triangular systems
    for (int i = k + 1; i < nt; i++) {
      #pragma omp task depend(in: a[k][k])
                       depend(inout: a[k][i])
      trsm(a[k][k], a[k][i], ts, ts);
    }

    // Update trailing matrix
    for (int i = k + 1; i < nt; i++) {
      for (int j = k + 1; j < i; j++) {
        #pragma omp task depend(inout: a[j][i])
                         depend(in: a[k][i], a[k][j])
        dgemm(a[k][i], a[k][j], a[j][i], ts, ts);
        }
        #pragma omp task depend(inout: a[i][i])
                         depend(in: a[k][i])
      syrk(a[k][i], a[i][i], ts, ts);
      }
    }
}
```

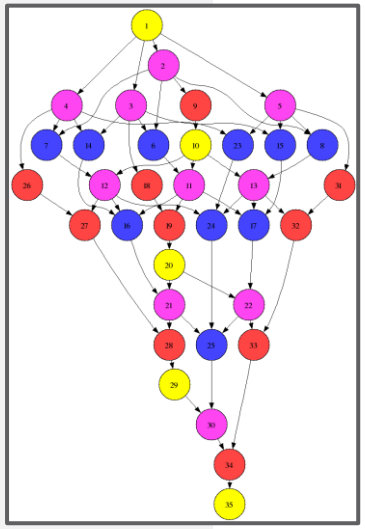


OpenMP 4.0

Motivation: Cholesky factorization

```
void cholesky(int ts, int nt, double* a[nt][nt]) {
  for (int k = 0; k < nt; k
    // Diagonal Block facto
    potrf(a[k][k], ts, ts);

    // Triangular systems
    for (int i = k + 1; i <
      #pragma omp task
      trsm(a[k][k], a[k][i]
    }
    #pragma omp taskwait

    // Update trailing matr
    for (int i = k + 1; i <
      for (int j = k + 1; j
        #pragma omp task
        dgemm(a[k][i], a[k]
      }
      #pragma omp task
      syrk(a[k][i], a[i][i]
    }
    #pragma omp taskwait
  }
}
```





OpenMP 4.0

Using 2017 Intel compiler

What's in the spec

What's in the Spec: syntax depend clause

```
depend([depend-modifier,] dependency-type: list-items)
```

where:

- `depend-modifier` is used to define iterators
- `dependency-type` may be: `in`, `out`, `inout`, `mutexinoutset` and `depobj`
- A `list-item` may be:
  - C/C++: A `lvalue` expr or an array section `depend(in: x, v[i], *p, w[10:10])`
  - Fortran: A variable or an array section `depend(in: x, v(i), w(10:20))`

What's in the Spec: depend clause (1)

- A task cannot be executed until all its predecessor tasks are completed

- If a task defines an `in` dependence over a list-item
  - → the task will depend on all previously generated sibling tasks that reference that list-item in an `out` or `inout` dependence

- If a task defines an `out`/`inout` dependence over list-item
  - → the task will depend on all previously generated sibling tasks that reference that list-item in an `in`, `out` or `inout` dependence

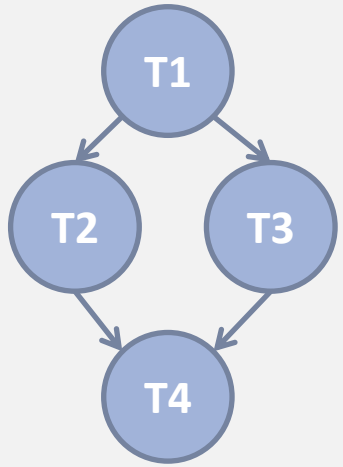
What's in the Spec: depend clause (1)

- A task cannot be executed until all its predecessor tasks are completed

- If a task defir
  - → the task will c ne of the list items in an `out` or `in`

- If a task defir
  - → the task will c ne of the list items in an `in, out` c

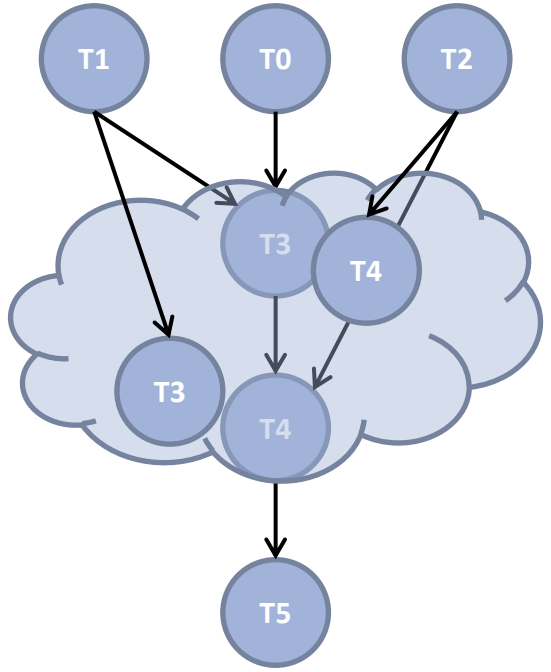
```
int x = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(inout: x) //T1
  { ... }

  #pragma omp task depend(in: x)    //T2
  { ... }

  #pragma omp task depend(in: x)    //T3
  { ... }

  #pragma omp task depend(inout: x) //T4
  { ... }
}
```



What's in the Spec: depend clause (2)

- New dependency type: `mutexinoutset`

```
int x = 0, y = 0, res = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(out: res)   //T0
   res = 0;

  #pragma omp task depend(out: x)   //T1
  long_computation(x);

  #pragma omp task depend(out: y)   //T2
  short_computation(y);

  #pragma omp task depend(in: x) depend(mutexinoutset: res) //T3
  res += x;

  #pragma omp task depend(in: y) depend(mutexinoutset: res) //T4
  res += y;

  #pragma omp task depend(in: res)   //T5
  std::cout << res << std::endl;
}
```



1. *inoutset property*: tasks with a `mutexinoutset` dependence create a cloud of tasks (an inout set) that synchronizes with previous & posterior tasks that dependent on the same list item

2. *mutex property*: Tasks inside the inout set can be executed in any order but with mutual exclusion

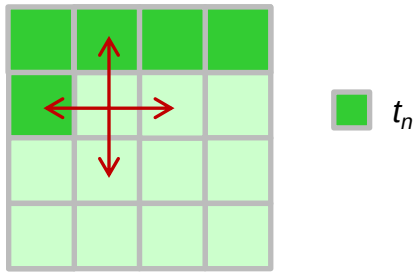
Use Case

Use Case: intro to Gauss-Seidel

```
void serial_gauss_seidel(int tsteps, int size, int (*p)[size]) {
  for (int t = 0; t < tsteps; ++t) {
    for (int i = 1; i < size-1; ++i) {
      for (int j = 1; j < size-1; ++j) {
        p[i][j] = 0.25 * (p[i][j-1] * // left
                          p[i][j+1] * // right
                          p[i-1][j] * // top
                          p[i+1][j]); // bottom
      }
    }
  }
}
```

Access pattern analysis

For a specific t, i and j

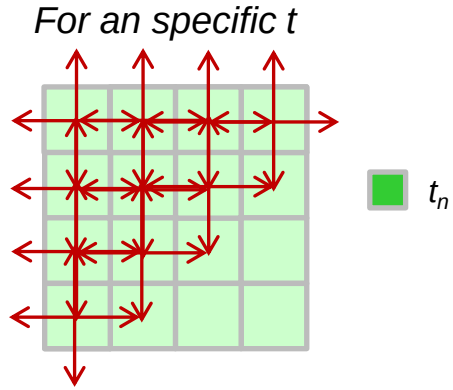


Each cell depends on:
- two cells (north & west) that are computed in the current time step, and
- two cells (south & east) that were computed in the previous time step

Use Case: Gauss-Seidel (2)

```
void serial_gauss_seidel(int tsteps, int size, int (*p)[size]) {
  for (int t = 0; t < tsteps; ++t) {
    for (int i = 1; i < size-1; ++i) {
      for (int j = 1; j < size-1; ++j) {
        p[i][j] = 0.25 * (p[i][j-1] * // left
                          p[i][j+1] * // right
                          p[i-1][j] * // top
                          p[i+1][j]); // bottom
      }
    }
  }
}
```

1st parallelization strategy



We can exploit the wavefront to obtain parallelism!!

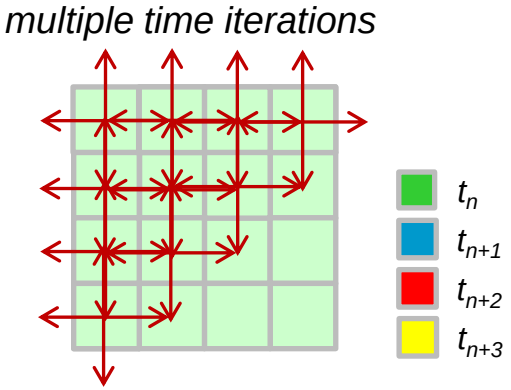
Use Case: Gauss-Seidel (3)

```
void gauss_seidel(int tsteps, int size, int TS, int (*p)[size]) {
  int NB = size / TS;
  #pragma omp parallel
  for (int t = 0; t < tsteps; ++t) {
    // First NB diagonals
    for (int diag = 0; diag < NB; ++diag) {
      #pragma omp for
      for (int d = 0; d <= diag; ++d) {
        int ii = d;
        int jj = diag - d;
         for (int i = 1+ii*TS; i < ((ii+1)*TS); ++i)
           for (int j = 1+jj*TS; i < ((jj+1)*TS); ++j)
             p[i][j] = 0.25 * (p[i][j-1] * p[i][j+1] *
                               p[i-1][j] * p[i+1][j]);
      }
    }
    // Lasts NB diagonals
    for (int diag = NB-1; diag >= 0; --diag) {
      // Similar code to the previous loop
    }
  }
}
```

Use Case: Gauss-Seidel (4)

```
void serial_gauss_seidel(int tsteps, int size, int (*p)[size]) {
  for (int t = 0; t < tsteps; ++t) {
    for (int i = 1; i < size-1; ++i) {
      for (int j = 1; j < size-1; ++j) {
        p[i][j] = 0.25 * (p[i][j-1] * // left
                          p[i][j+1] * // right
                          p[i-1][j] * // top
                          p[i+1][j]); // bottom
      }
    }
  }
}
```

2nd parallelization strategy

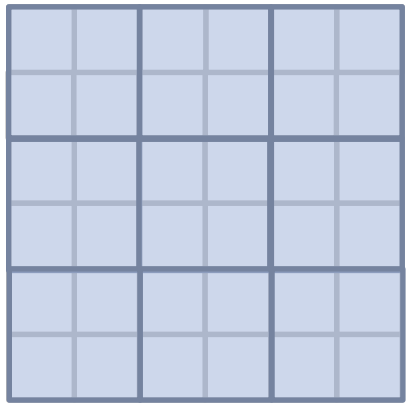


We can exploit the wavefront of multiple time steps to obtain MORE parallelism!!

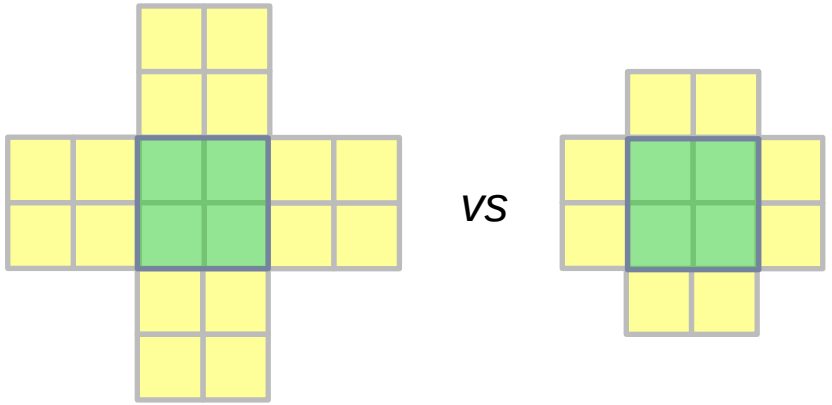
Use Case: Gauss-Seidel (5)

```
void gauss_seidel(int tsteps, int size, int TS, int (*p)[size]) {
  int NB = size / TS;

  #pragma omp parallel
  #pragma omp single
  for (int t = 0; t < tsteps; ++t)
    for (int ii=1; ii < size-1; ii+=TS)
      for (int jj=1; jj < size-1; jj+=TS) {
        #pragma omp task depend(inout: p[ii:TS][jj:TS])
            depend(in: p[ii-TS:TS][jj:TS], p[ii+TS:TS][jj:TS],
                       p[ii:TS][jj-TS:TS], p[ii:TS][jj:TS])
        {
          for (int i=ii; i<(1+ii)*TS; ++i)
            for (int j=jj; j<(1+jj)*TS; ++j)
               p[i][j] = 0.25 * (p[i][j-1] * p[i][j+1] *
                                 p[i-1][j] * p[i+1][j]);
        }
      }
}
```

inner matrix region

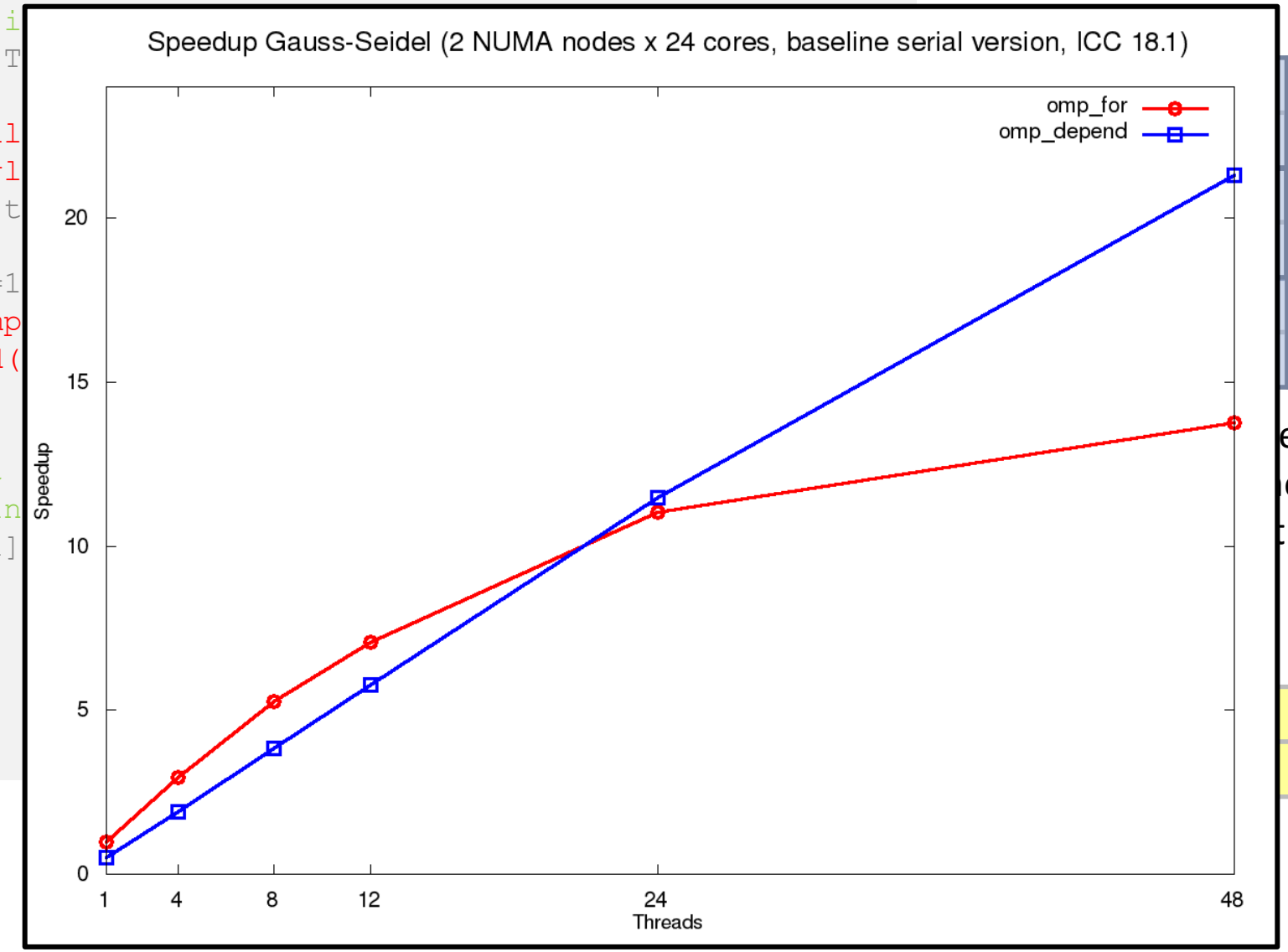


Q: Why do the input dependences depend on the whole block rather than just a column/row?

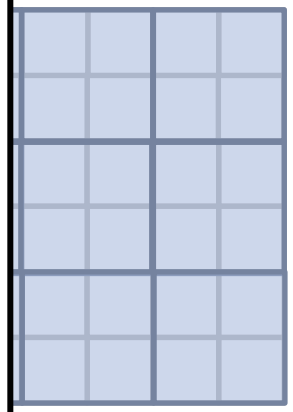


Use Case: Gauss-Seidel (5)

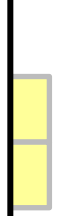
```
void gauss_seidel(i
  int NB = size / T

  #pragma omp paral
  #pragma omp singl
  for (int t = 0; t
    for (int ii=1;
      for (int jj=1
        #pragma omp
            depend(


        {
          for (int
            for (in
              p[i]

        }
      }
}
```



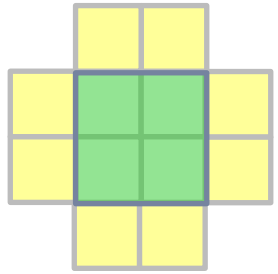
matrix region



e input dependences
e whole block rather
t a column/row?



vs



Summary

- Understand the NUMA nature of the system and take that into account when parallelizing with OpenMP
- You can employ affinity in your code to improve performance
- SIMD can bring a performance benefit via an additional parallel model inside OpenMP to leverage vector units in the processor
- Advanced tasking patterns can help gain additional performance of OpenMP tasking

Thanks for your attention!