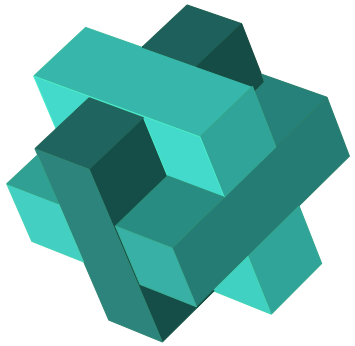
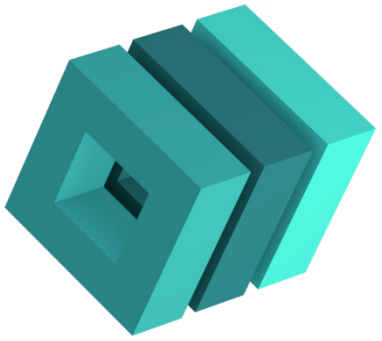
Advanced OpenMP® Tutorial



OpenMPCon
DEVELOPERS CONFERENCE

IWOMP
INTERNATIONAL WORKSHOP



Credits, Credits, Credits

Presented by

Michael Klemm (OpenMP CEO)

Xavier Martorell

Xavier Teruel

Slides developed by members of the OpenMP Language Committee

Christian Terboven
Michael Klemm
Ruud van der Pas
Eric Stotzer
Bronis R. de Supinski
Sergi Mateo

Agenda

Topic	Speaker	Time
What is “Advanced OpenMP?”/Miscellaneous features	Xavier M.	10 min
OpenMP Tasking	Xavier T.	50 min
NUMA Awarenesss	Xavier M.	20 min
Vectorization/SIMD	Michael	40 min

Updated Slides

http://bit.ly/201809tutorial

What is “Advanced OpenMP”?

What is “Advanced OpenMP”?

Multiple choice:

1. All the things that you may have heard of, but have never used…
2. Things which have appeared in OpenMP since you took that undergraduate course
3. Anything beyond `!$omp parallel for`
4. All of the above

All of the above is a good answer. We may not be able to cover it all, though!

Recent OpenMP Features

Major:

- Tasking (coming up soon, Xavier T.)
- NUMA awareness (Xavier M.)
- Vectorization (Michael)
- Offload to accelerator devices (not covered today)

Minor (next, small, simple, give you time to wake up ☺)

- Lock/critical/atomic (5.0) hints
- New dynamic schedule

Lock/critical/atomic hints What?

A way of giving the implementation more information about the way you'd like a lock or critical section to be implemented

A new lock initialization function `omp_init_lock_with_hint(…)`

A hint clause on `omp critical` (and, in 5.0 `omp atomic`)

A set of synchronization hints

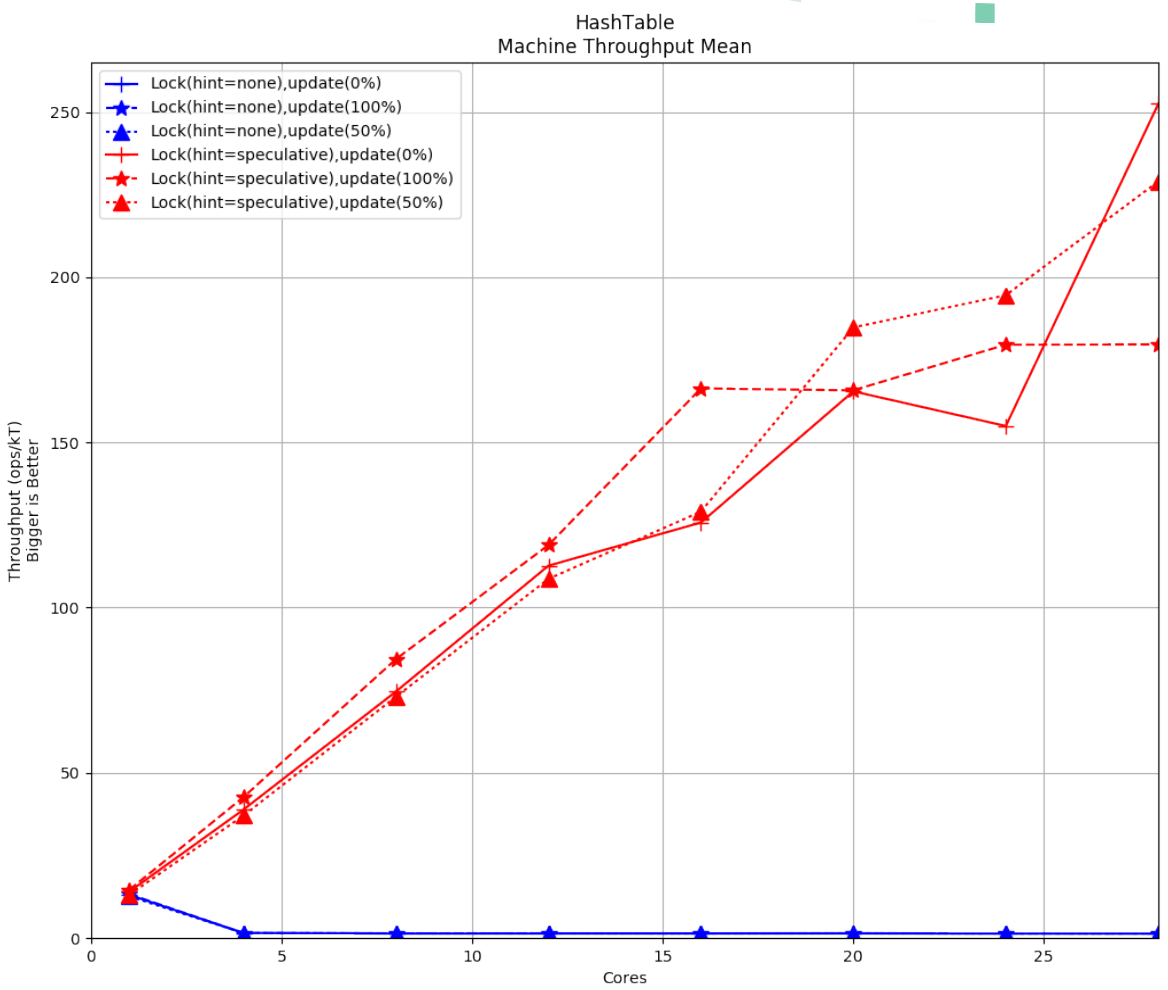
```
omp_sync_hint_{none, contended/uncontended,
                speculative/nonspeculative}
```

Lock/critical/atomic hints Why?

Modern processors support speculative execution (“transactional memory”). Present in processors from Intel, IBM, …

Allows concurrent execution of critical sections if they do not conflict

Can give the performance of a fine-grained reader/writer lock while only having to code a simple coarse grained lock



Experiment details

Take std::unordered_map<uint32_t,uint32_t> and wrap it in OpenMP locks.

```
class lockedHash {
    std::unordered_map<uint32_t, uint32_t> theMap;
    omp_lock_t theLock;
public:
    lockedHash(omp_lock_hint_t hint) {omp_init_lock_with_hint(&theLock, hint);}

    void insert(uint32_t key, uint32_t value) {
        omp_set_lock(&theLock);
        theMap.insert({key,value});
        omp_unset_lock(&theLock);
    }

    uint32_t lookup(uint32_t key) {
        omp_set_lock(&theLock);
        auto result = theMap.find(key);
        omp_unset_lock(&theLock);
        return result == theMap.end() ? 0 : result->second;
    }
};
```

Only change lock initialization…
No changes to lock use

Measure total machine throughput as we add cores (1T/C), doing lookups or updates as fast as they can when using `omp_sync_hint_none` and `omp_sync_hint_speculative` to initialize `theLock`.

New dynamic scheduling option

`schedule({monotonic,nonmonotonic}:dynamic)`

`nonmonotonic` allows an iteration stealing scheduling scheme which can out-perform a default dynamic schedule.

Beware: `nonmonotonic` is becoming the default schedule in OpenMP 5.0

Difference: monotonic requires each thread sees iterations which only move in one direction, nonmonotonic allows them to move "backwards"

e.g. in `for(i=0; i<5; i++)` a thread may see `3,4,0` with a `nonmonotonic:dynamic` schedule.

Agenda

Topic	Speaker	Time
What is "Advanced OpenMP?"/Miscellaneous features	Xavier M.	10 min
OpenMP Tasking	Xavier T.	50 min
NUMA Awarenesss	Xavier M.	20 min
Vectorization/SIMD	Michael	40 min

OpenMP Tasking

Irregular and Recursive Parallelism

OpenMP Tasking

- What is tasking?
- Introduction by Example: Sudoku
- Data Scoping
- Scheduling and Dependencies
- Taskloops
- More Tasking Stuff

What **is** tasking?
First: What is “Classic” OpenMP?

“Classic” OpenMP treats threads as a fundamental concept

- You know how many there are (`omp_get_num_threads()`)
- You know which one you are (`omp_get_thread_num()`)
- A major concern is how to share work between threads
  - Choice of `schedule` clause on for loops
  - Explicit decisions based on `omp_get_thread_num()`
  - A whole section in the standard on Worksharing Constructs!
- The standard describes semantics in terms of threads, e.g. for barrier

  “All threads of the team executing the binding **parallel** region must execute the **barrier** region…”

What is tasking?
Task model

Tasking lifts your thinking

- Forget about threads, and about scheduling work to them
- Instead think how your code can be broken into chunks of work which can execute in parallel (“tasks”)
- Let the runtime system handle how to execute the work
  - We’re not going to discuss how this works, but it is fun. Talk to me if you want to find out more.
- Think in terms of work being complete rather than threads getting to some point in the code
- Ideas from Cilk, also implemented in TBB for C++

Problems with traditional worksharing

- Worksharing constructs do not compose well
- Pathological example: parallel dgemm

```
void example() {
#pragma omp parallel
    {
        compute_in_parallel(A);
        compute_in_parallel_too(B);
        // dgemm is either parallel or sequential,
        // but has no orphaned worksharing
        cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                    m, n, k, alpha, A, k, B, n, beta, C, n);

}   }
```

- Writing such code either
  - oversubscribes the system,
  - yields bad performance due to OpenMP overheads, or
  - needs a lot of glue code to use sequential dgemm only for sub-matrixes

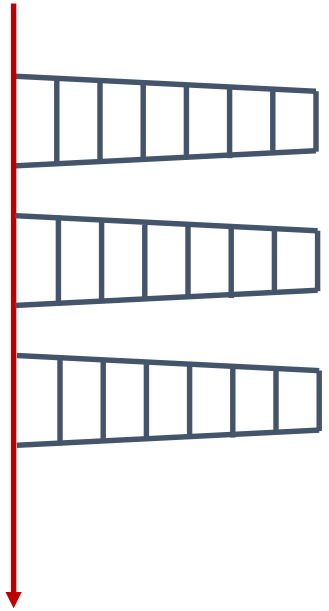
Ragged Fork/Join

- Traditional worksharing can lead to ragged fork/join patterns

```
void example() {
    compute_in_parallel(A);

    compute_in_parallel_too(B);

    cblas_dgemm(…, A, B, …);
}
```



Introduction by Example: Sudoku

Let‘s solve Sudoku puzzles with brute multi-core force

Find an empty cell
For value in 0:15
    If (not valid) continue
    Recurse for next empty cell or print result
    if this was the last cell.
Wait for completion

Note: this is a 16x16 sudoku so we‘re seaching ~16^{220} = 8.e264 configurations!

	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	

Why Do We Need Tasks?

This is a recursive problem

Tasks will take different amounts of time

- Some rapidly reach an inconsistent state
- Some nearly succeed, so run for much longer
- One succeeds (assuming a well defined problem!)

We want to exploit parallelism at every level

- But nested OpenMP parallelism is “complicated” ☺

The OpenMP Task Construct

C/C++

```
#pragma omp task [clause]
... structured block ...
```

Fortran

```
!$omp task [clause]
... code ...
!$omp end task
```

Each encountering thread/task creates a new task

- Code and data is packaged up
- Tasks can be nested
  - Into another task directive
  - Into a Worksharing construct

Data scoping clauses:

- `shared`(*list*)
- `private`(*list*) `firstprivate`(*list*)
- `default`(*shared* | *none*)

Barrier and Taskwait Constructs

OpenMP `barrier` (implicit or explicit)

- All tasks created by any thread of the current *Team* are guaranteed to have completed at barrier exit

C/C++

```
#pragma omp barrier
```

Fortran

```
!$omp barrier
```

Task barrier: `taskwait`

- Encountering task is suspended until child tasks complete
  - Applies only to children, not all descendants!

C/C++

```
#pragma omp taskwait
```

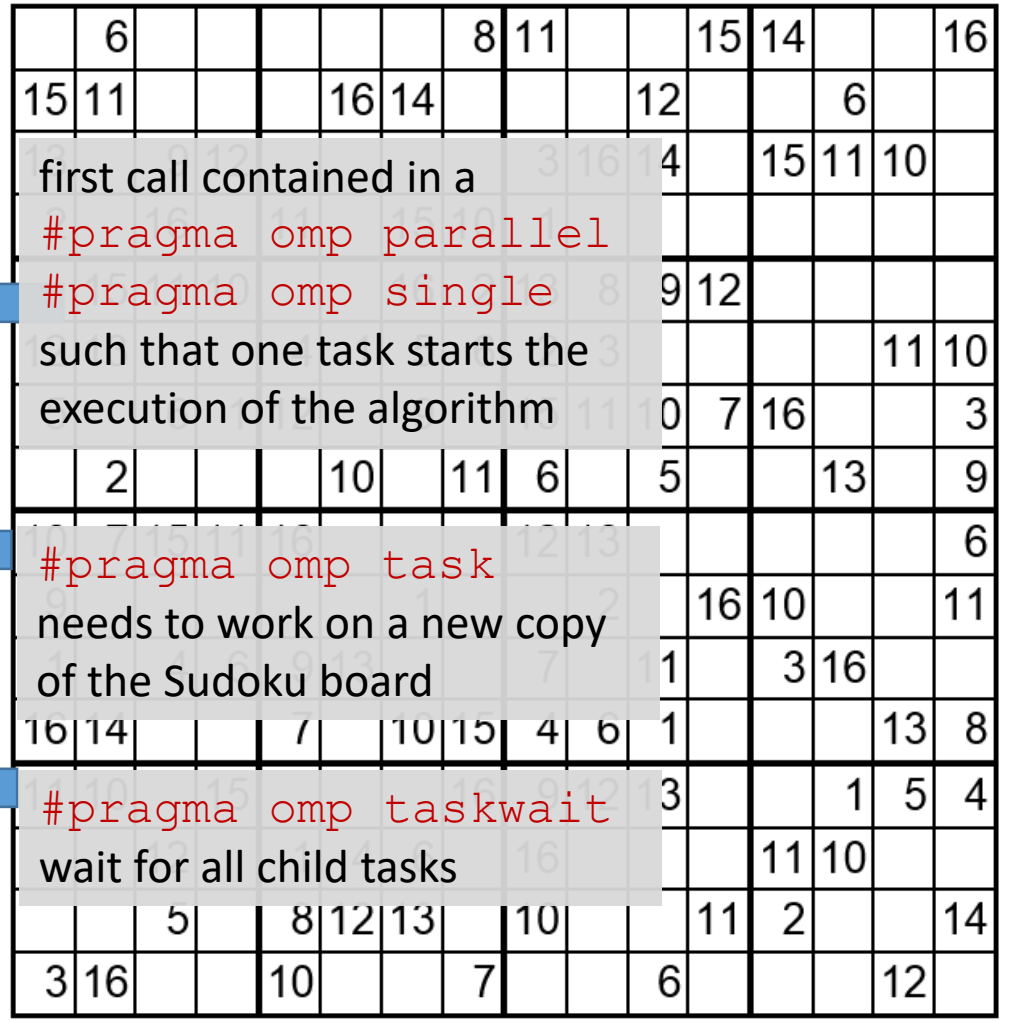
Fortran

```
!$omp taskwait
```

Parallel Brute-force Sudoku

This parallel algorithm finds all valid solutions
Find an empty cell
For value in 0:15
    If (not valid) continue
    Recurse for next empty cell,or print result

Wait for completion



Parallel Brute-force Sudoku (2/3)

OpenMP parallel region creates a team of threads

```
#pragma omp parallel
{
#pragma omp single
   solve_parallel(0, 0, sudoku2,false);
} // end omp parallel
```

- Single construct: One thread enters the execution of `solve_parallel`
- the other threads wait at the end of the `single` …
  - … and are ready to pick up tasks from the work queue

Parallel Brute-force Sudoku (3/3)

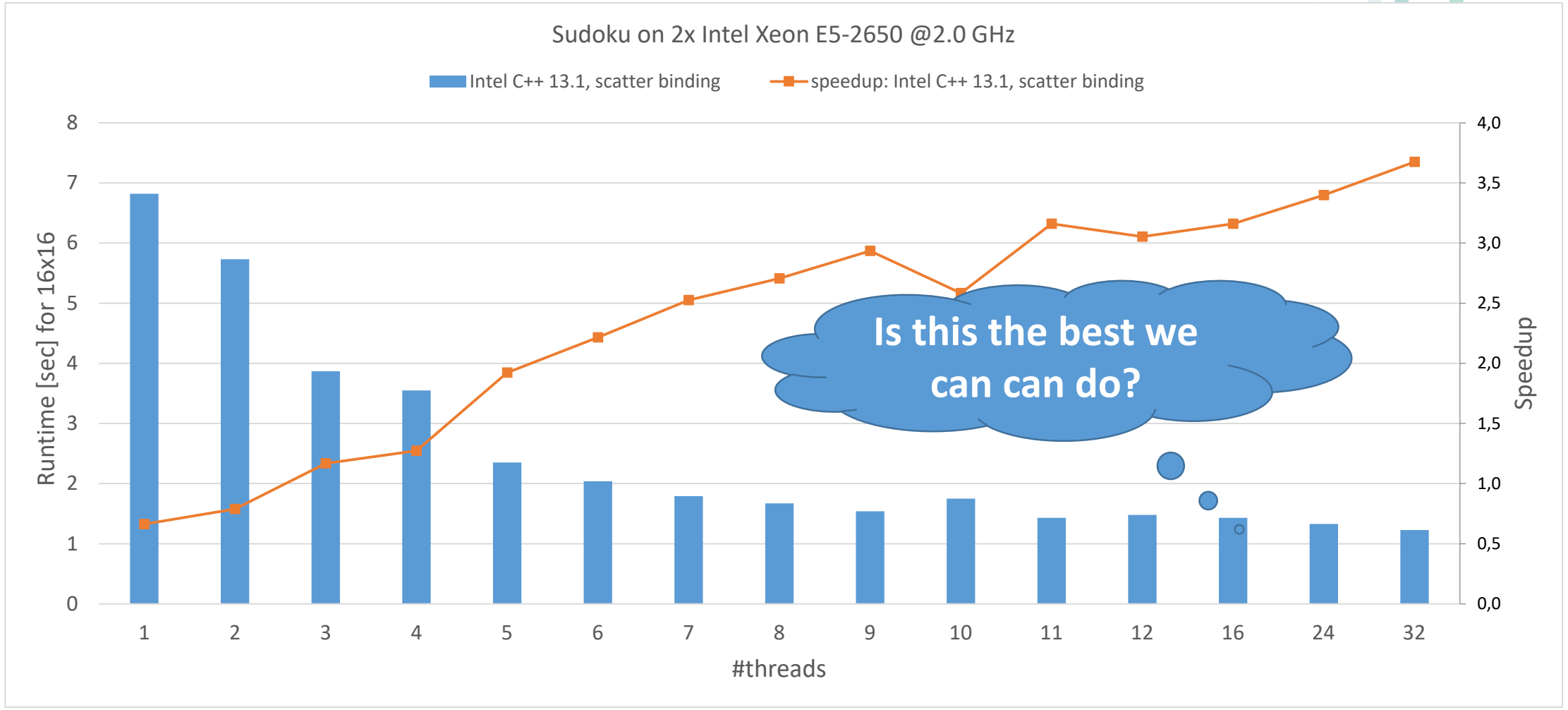
The actual implementation

```
for (int i = 1; i <= sudoku->getFieldSize(); i++) {
   if (!sudoku->check(x, y, i)) {
#pragma omp task firstprivate(i,x,y,sudoku)
      {
        // create from copy constructor
        // CSudokuBoard new_sudoku(*sudoku);
        sudoku.set(y, x, i);
        if (solve_parallel(x+1, y, &sudoku))
        sudoku.printBoard();
      } // end omp task
   }
}
#pragma omp taskwait
```

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

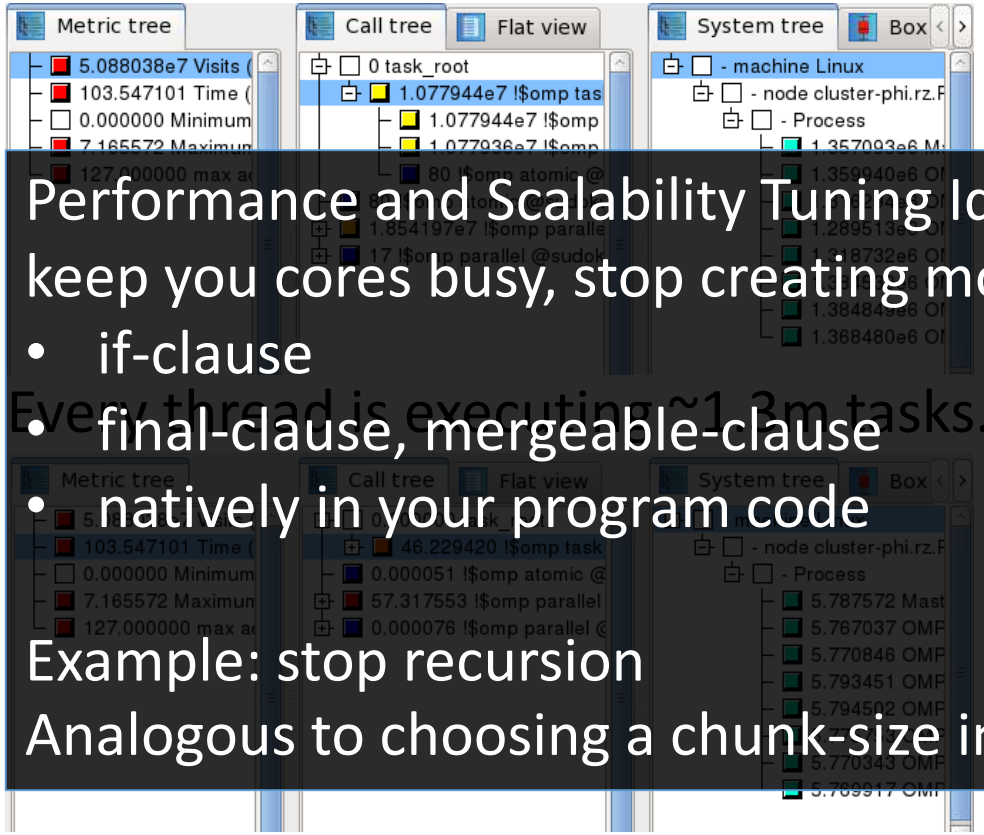
`#pragma omp taskwait`
wait for all child tasks

Performance Evaluation

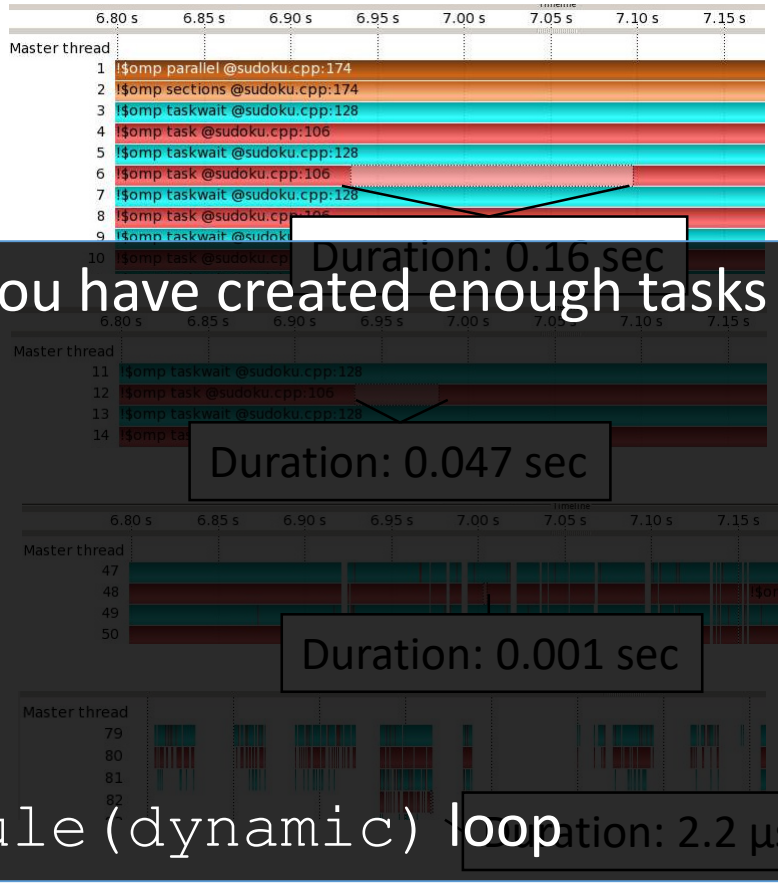


Performance Analysis

Event-based profiling gives a good overview :



Tracing gives more details:



Performance and Scalability Tuning Idea: when you have created enough tasks to keep you cores busy, stop creating more tasks!

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

Example: stop recursion

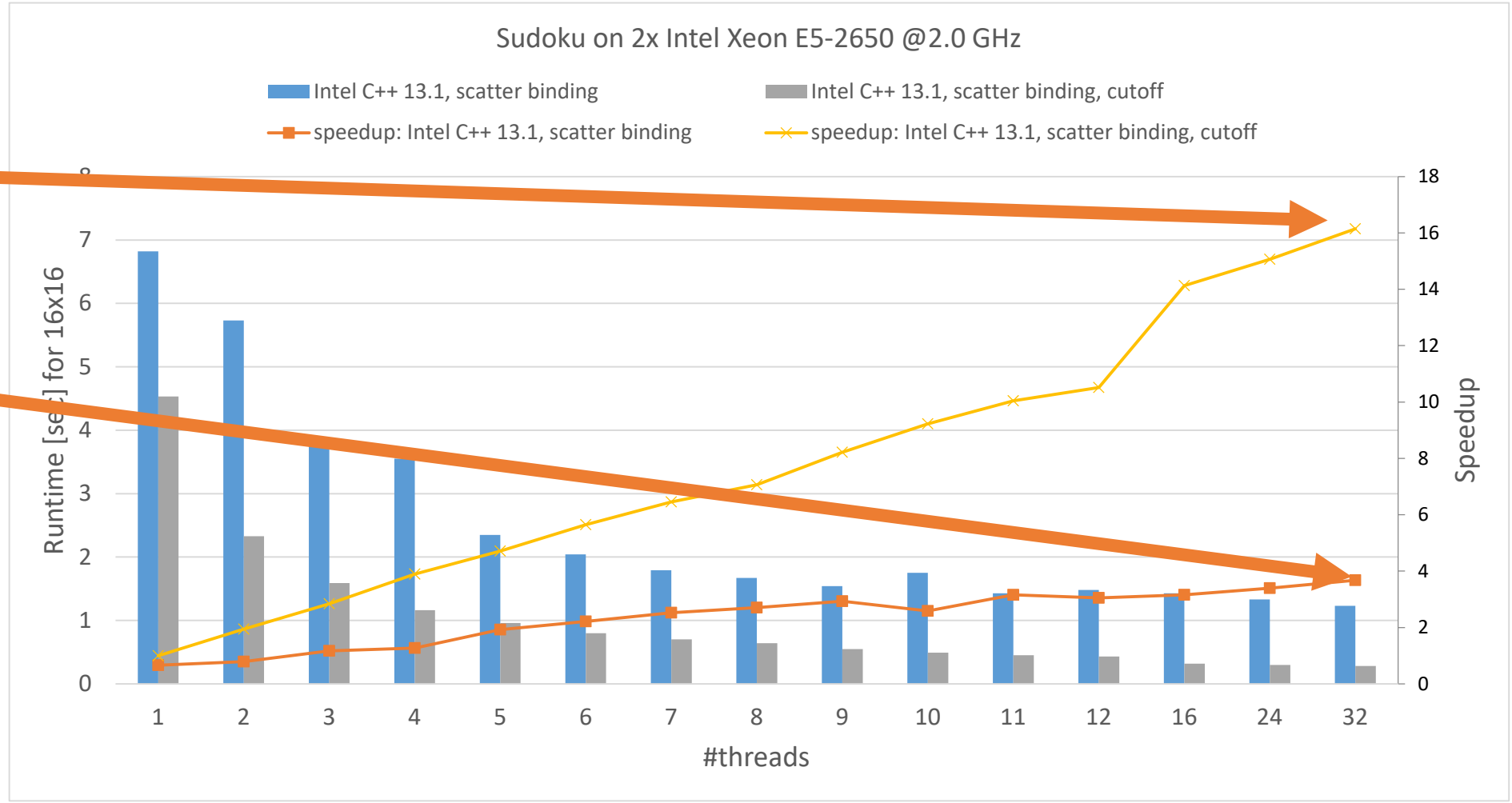
Analogous to choosing a chunk-size in a `schedule(dynamic)` loop

… in ~5.7 seconds => average duration of a task is ~4.4 µs

Tasks get much smaller down the call-stack.

Performance Evaluation

Now have >16x speedup where we had <4x before!



Task Data Scoping

Some rules from *Parallel Regions* apply:

- Static and Global variables are shared
- Automatic Storage (local) variables are private

If `shared` scoping is not inherited:

- Orphaned Task variables are `firstprivate` by default!
- Non-Orphaned Task variables inherit the `shared` attribute!
- → Variables are `firstprivate` unless `shared` in the enclosing context

Data Scoping Example

```
int a = 1;
void foo()
{
   int b = 2, c = 3;
   #pragma omp parallel private(b)
   {
        int d = 4;
        #pragma omp task
        {
               int e = 5;
               // Scope of a: shared          value of a: 1
               // Scope of b: firstprivate    value of b: undefined (Why? ☺)
               // Scope of c: shared          value of c: 3
               // Scope of d: firstprivate    value of d: 4
               // Scope of e: private         value of e: 5
        }
    }
}
```

Use `default(none)`!

```
int a = 1;
void foo()
{
   int b = 2, c = 3;
   #pragma omp parallel private(b)
   {
        int d = 4;
        #pragma omp task
        {
               int e = 5;
               // Scope of a: shared,          value of a: 1
               // Scope of b: firstprivate,    value of b: undefined
               // Scope of c: shared,          value of c: 3
               // Scope of d: firstprivate,    value of d: 4
               // Scope of e: private,         value of e: 5
        }
    }
}
```

Hint: Use `default(none)` to be forced to think about every variable if the scope is not obvious

Scheduling

- Default: Tasks are *tied* to the thread that first executes them this is normally not the creator. Scheduling constraints:
  - Only the thread to which a task is tied can execute it
  - A task can only be suspended at task scheduling points
    - Task creation, task finish, `taskwait`, `barrier`, `taskyield`
  - If task is not suspended in a barrier, the executing thread can only switch to a direct descendant of a task tied to the thread
- Tasks created with the `untied` clause are never tied
  - Allowed to resume at task scheduling points in a different thread
  - ~~No scheduling restrictions, e.g., can be suspended at any point~~
  - Gives more freedom to the implementation, e.g., load balancing

Unsafe use of `untied` Tasks

- Problem: Because untied tasks may migrate between threads at any point, thread-centric constructs can yield unexpected results
- Remember when using `untied` tasks:
  - Avoid `threadprivate` variables
  - Avoid any use of thread-ids (i.e., `omp_get_thread_num()`)
  - Be careful with `critical region` and *locks*

Good advice anyway!

`if` Clause

- When the expression in an `if` clause on a task evaluates to `false`
  - The encountering task is suspended
  - The new task is executed immediately
  - The parent task resumes when the new task finishes

  → Used for optimization, e.g., avoid creation of small tasks

The `taskyield` Directive

C/C++	Fortran
`#pragma omp taskyield`	`!$omp taskyield`

- The `taskyield` directive specifies that the current task can be suspended in favour of execution of a different task.
  - **Hint** to the runtime for optimization and/or deadlock prevention
  - But, since it‘s only a hint it can be ignored, so you cannot rely on it to prevent deadlock

taskyield Example (1/2)

```
#include <omp.h>

void something_useful();
void something_critical();

void foo(omp_lock_t * lock, int n)
{
    for(int i = 0; i < n; i++)
        #pragma omp task
        {
            something_useful();
            while( !omp_test_lock(lock) ) {
                #pragma omp taskyield
            }
            something_critical();
            omp_unset_lock(lock);
        }
}
```

Taskyield allows the spinning task to be suspended here, letting the executing thread perform other work.

`priority` Clause

C/C++

```
#pragma omp task priority(priority-value)
... structured block ...
```

Fortran

```
!$omp task priority(priority-value)
...
!$omp end task
```

- The *priority* is a **hint** to the runtime system for task execution order
- Among all tasks ready to be executed, higher priority tasks are recommended to execute before lower priority ones
  - priority is non-negative numerical scalar (default: 0)
  - priority <= max-task-priority ICV
    - environment variable OMP_MAX_TASK_PRIORITY
- You **cannot** rely on task execution order being determined by this clause; it‘s only a hint and can be ignored!

`final` Clause

C/C++

```
#pragma omp task final(expr)
```

Fortran

```
!$omp task final(expr)
```

- For recursive problems that perform task decomposition, stopping task creation at a certain depth exposes enough parallelism but reduces overhead.
- Beware: merging the data environment may have side-effects

```
void foo(bool arg)
{
   int i = 3;
   #pragma omp task final(arg) firstprivate(i)
      i++;                    // No externally visible effect if in task...
   printf("%d\n", i);         // Could print 3 or 4 depending on arg
}
```

`mergeable` Clause

C/C++

```
#pragma omp task mergeable
```

Fortran

```
!$omp task mergeable
```

- If the `mergeable` clause is present, the implementation is allowed to merge the task's data environment
  - if the generated task is undeferred or included
    - undeferred: if clause present and evaluates to false
    - included: final clause present and evaluates to true
- As far as I know, no compiler or runtime exploits `final` or `mergeable` so using them is currently futile (other than to provide evidence to use to hassle your compiler vendor ☺)

The `taskgroup` Construct

C/C++

```
#pragma omp taskgroup
... structured block ...
```

Fortran

```
!$omp taskgroup
... structured block ...
!$omp end task
```

- Specifies a wait for completion of child tasks **and their descendant tasks**
  - This is deeper sychronization than `taskwait`, but
  - with the option to restrict to a subset of all tasks (as opposed to a `barrier`)

Task Dependencies: Motivation

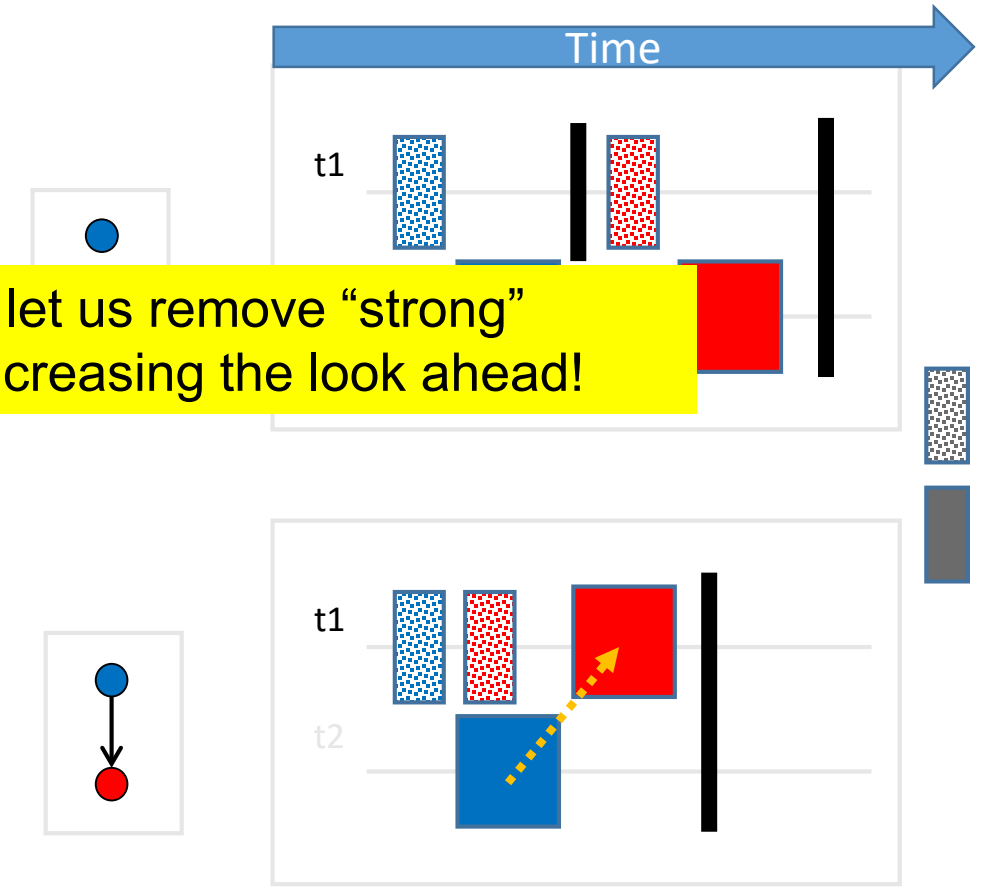
- Task dependences are a way to define task-execution constraints

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)
  std::cout << x << std::endl;
  #pragma omp task depend(inout: x)
  x++;
}
```



Task dependencies let us remove "strong" synchronizations, increasing the look ahead!

Controlling when a task starts

- In more complicated codes we have dependencies between tasks
- For instance, suppose one task(*b*) cannot start until another(*a*) has finished because *b* needs to consume data which was written by *a*
- OpenMP provides task dependencies to let you express these constraints
  - `depend(in:`*`var`*`)` => this task consumes *var*
  - `depend(out:`*`var`*`)` => this task produces *var*
  - `depend(inout:`*`var`*`)` => this task consumes *var* and updates it

  Coming in OpenMP 5.0
  - `depend(mutexinoutset:`*`var`*`)` only one task using *var* can run at a time

The depend Clause

C/C++

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

Fortran

```
!$omp task depend(dependency-type: list)
... code ...
!$omp end task
```

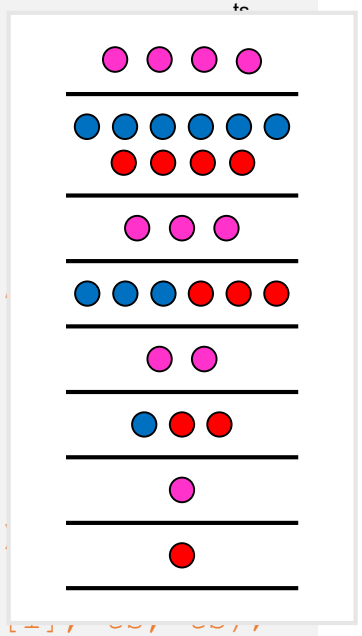
- The *task dependence* is fulfilled when the predecessor task has completed
  - `in` dependency-type: the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `out` or `inout` clause.
  - `out` and `inout` dependency-type: 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.
  - `mutexinoutset`: only one task in the set may execute at any time (OpenMP 5.0!)
  - The list items in a `depend` clause may include array sections.

Example: 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], ts, ts);
      }
      #pragma omp task
      syrk(a[k][i], a[i][i], ts, ts);
    }
    #pragma omp taskwait
  }
}
```

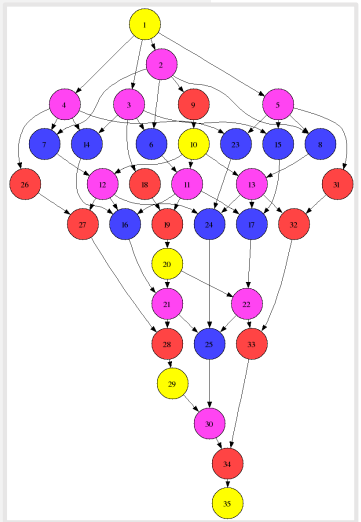


OpenMP 3.1

```
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);
    }
  }
}
```

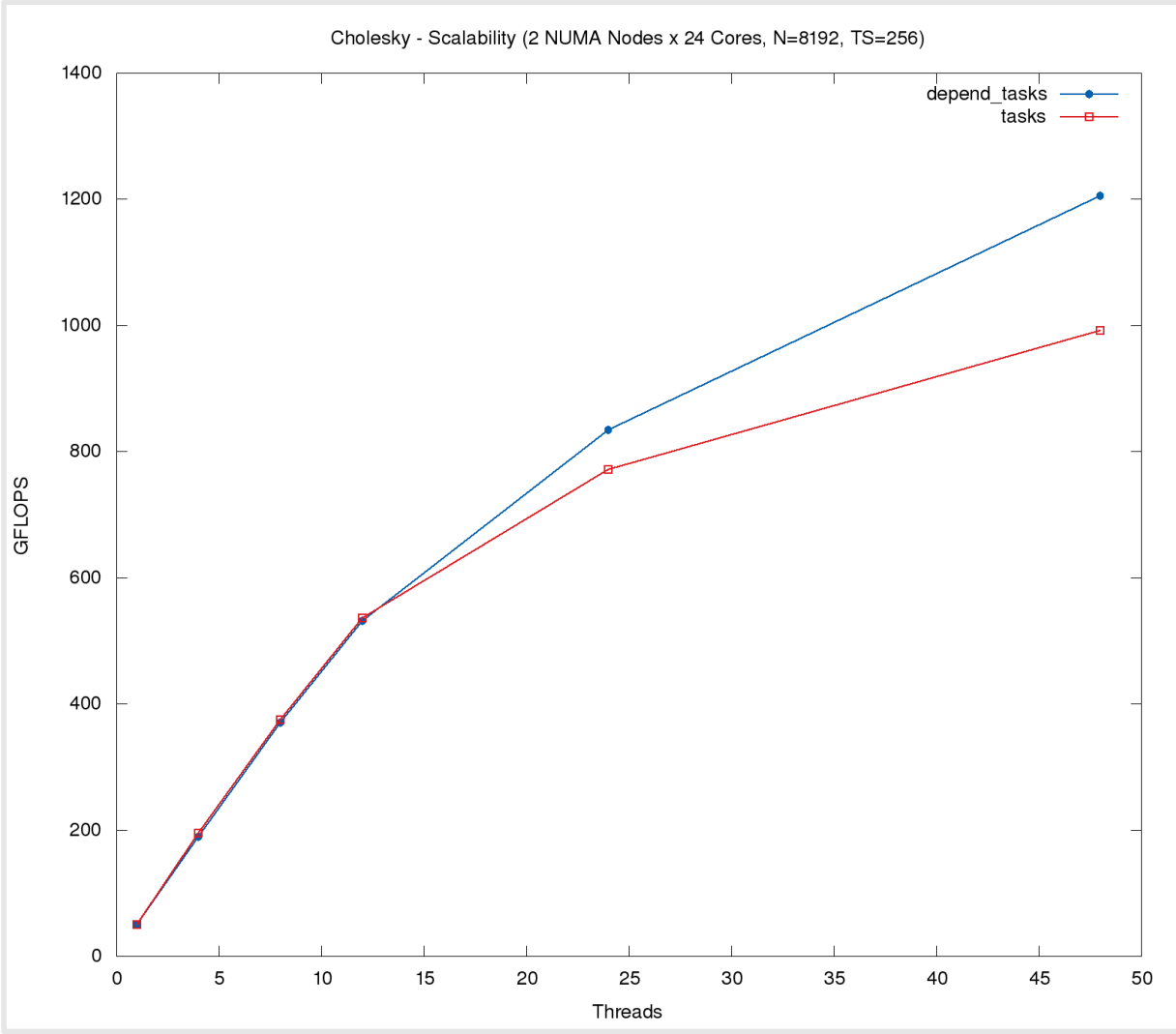


OpenMP 4.0

Example: Cholesky factorization

Jack Dongarra on OpenMP Task Dependencies:
[…] The appearance of DAG scheduling constructs in the OpenMP 4.0 standard offers a particularly important example of this point. Until now, libraries like PLASMA had to rely on custom built task schedulers; […] However, the inclusion of DAG scheduling constructs in the OpenMP standard, along with the rapid implementation of support for them [...], throws open the doors to widespread adoption of this model in academic and commercial applications for shared memory. **We view OpenMP as the natural path forward for the PLASMA library and expect that others will see the same advantages to choosing this alternative.**

Full article
here: http://www.hpcwire.com/2015/10/19/numerical-algorithms-and-libraries-at-exascale/



Using 2017 Intel compiler

The `taskloop` Construct

- Parallelize a loop using OpenMP tasks
  - Cut loop into chunks
  - Create a task for each loop chunk

- Syntax (C/C++)

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

- Syntax (Fortran)

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

Clauses for `taskloop` Construct

- Taskloop construct inherits clauses both from worksharing constructs and the `task` construct
  - `shared`, `private`
  - `firstprivate`, `lastprivate`
  - `default`
  - `collapse`
  - `final`, `untied`, `mergeable`

- `grainsize(`*`grain-size`*`)`
  Chunks have at least *grain-size* and max 2**grain-size* loop iterations

- `num_tasks(num-tasks)`
  Create *num-tasks* tasks for iterations of the loop

Example: Sparse CG

```
for (iter = 0; iter < sc->maxIter; iter++)
{
    precon(A, r, z);
    vectorDot(r, z, n, &rho);
    beta = rho / rho_old;
    xpay(z, beta, n, p);
    matvec(A, p, q);
    vectorDot(p, q, n, &dot_pq);
    alpha = rho / dot_pq;
    axpy(alpha, p, n, x);
    axpy(-alpha, q, n, r);
    sc->residual = sqrt(rho) * bnrm2;
    if (sc->residual <= sc->tolerance)
        break;
    rho_old = rho;
}
```

```
void matvec(Matrix *A, double *x, double *y) {
    // ...
#pragma omp parallel for \
            private(i,j,is,ie,j0,y0) \
            schedule(static)
for (i = 0; i < A->n; i++) {
        y0 = 0;
        is = A->ptr[i];
        ie = A->ptr[i + 1];
        for (j = is; j < ie; j++) {
            j0 = index[j];
            y0 += value[j] * x[j0];
        }
        y[i] = y0;
    }
    // ...
}
```

Example: Sparse CG

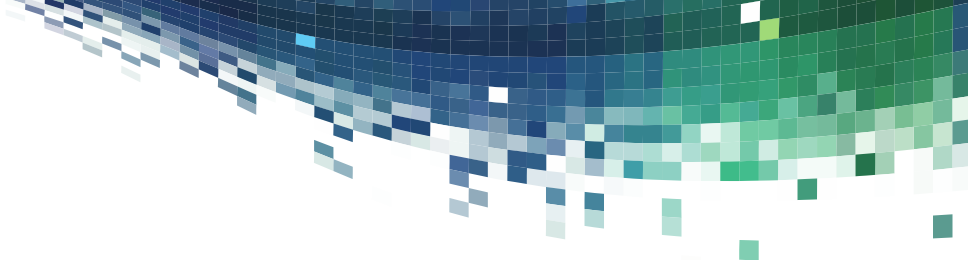
```
#pragma omp parallel
#pragma omp single
for (iter = 0; iter < sc->maxIter; iter++)
{
    precon(A, r, z);
    vectorDot(r, z, n, &rho);
    beta = rho / rho_old;
    xpay(z, beta, n, p);
    matvec(A, p, q);
    vectorDot(p, q, n, &dot_pq);
    alpha = rho / dot_pq;
    axpy(alpha, p, n, x);
    axpy(-alpha, q, n, r);
    sc->residual = sqrt(rho) * bnrm2;
    if (sc->residual <= sc->tolerance)
        break;
    rho_old = rho;
}
```

```
void matvec(Matrix *A, double *x, double *y) {
    // ...

#pragma omp taskloop private(j,is,ie,j0,y0) \
            grain_size(500)
    for (i = 0; i < A->n; i++) {
        y0 = 0;
        is = A->ptr[i];
        ie = A->ptr[i + 1];
        for (j = is; j < ie; j++) {
            j0 = index[j];
            y0 += value[j] * x[j0];
        }
        y[i] = y0;
    }
    // ...
}
```

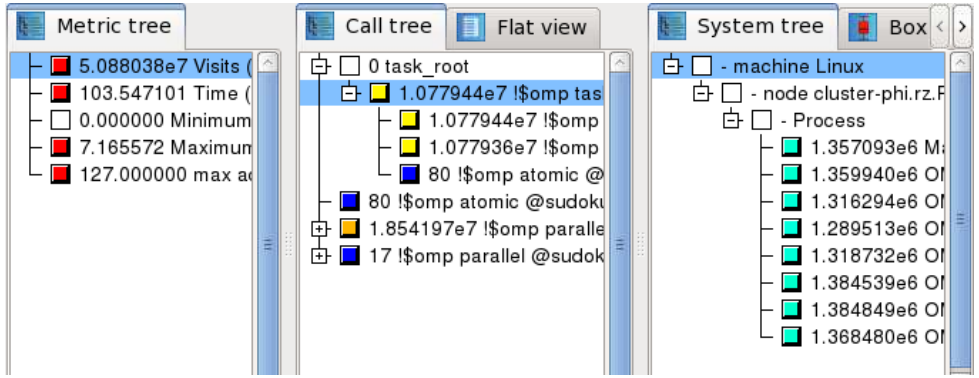
Conclusions

- Tasking allows you
  - to exploit recursive parallelism which is hard to do with classic worksharing
  - to exploit parallelism in places where there are complicated data-flow dependences between computations
  - to go beyond threads



Performance Analysis

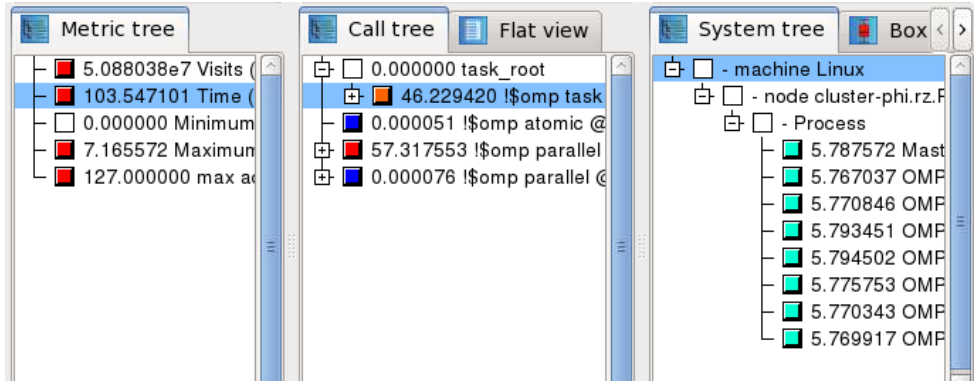
Event-based profiling gives a good overview :



lvl 6

lvl 12

Every thread is executing ~1.3m tasks…

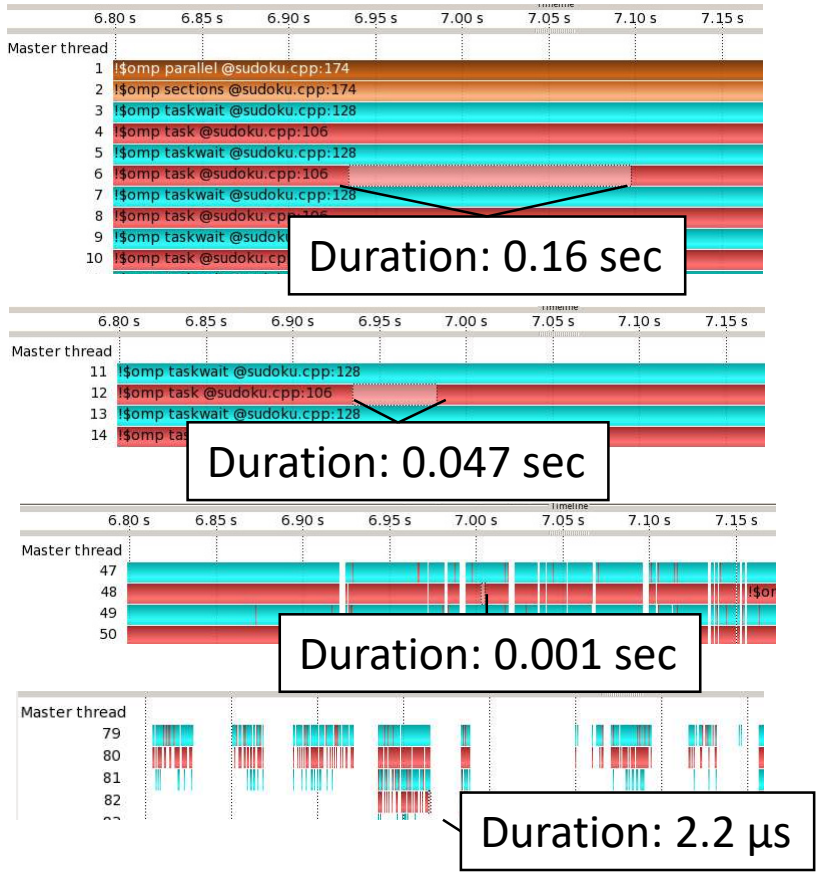


lvl 48

lvl 82

… in ~5.7 seconds => average duration of a task is ~4.4 µs

Tracing gives more details:



Tasks get much smaller down the call-stack.

NUMA Awareness

Or: A Game of Latency and Bandwidth

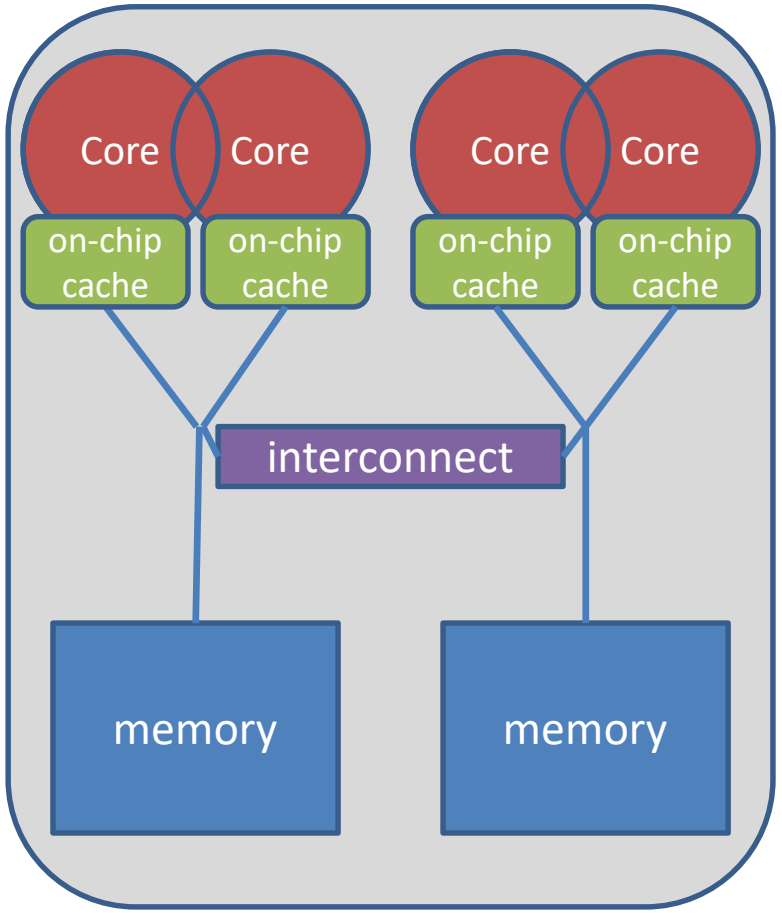
OpenMP and Performance

- Two of the more obscure things that can negatively impact performance are cc-NUMA effects and false sharing

- ***Neither of these are inherent to OpenMP, but to shared-memory parallel programming***
  - But they most show up because you used OpenMP
  - They are important enough so that OpenMP worries about them

Non-uniform Memory

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

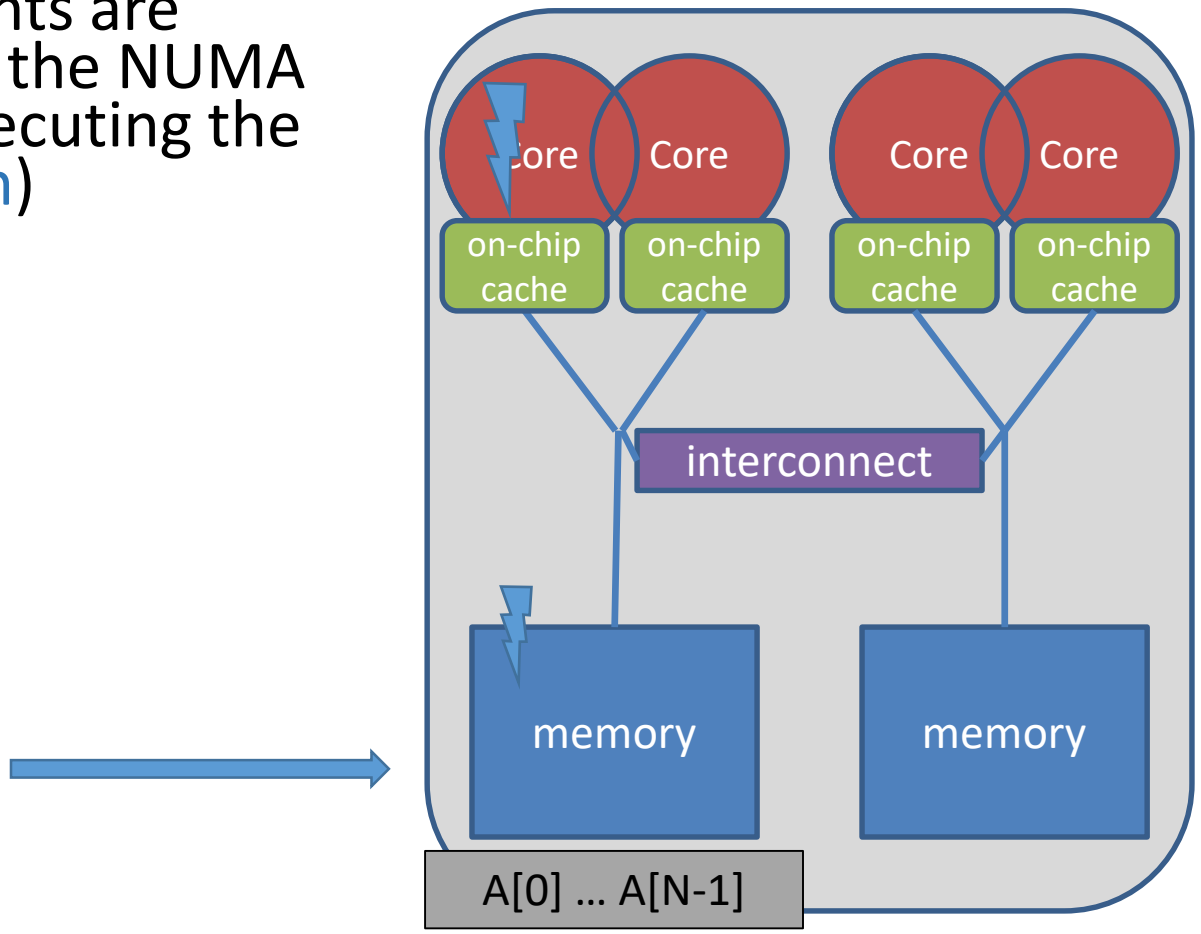


Non-uniform Memory

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

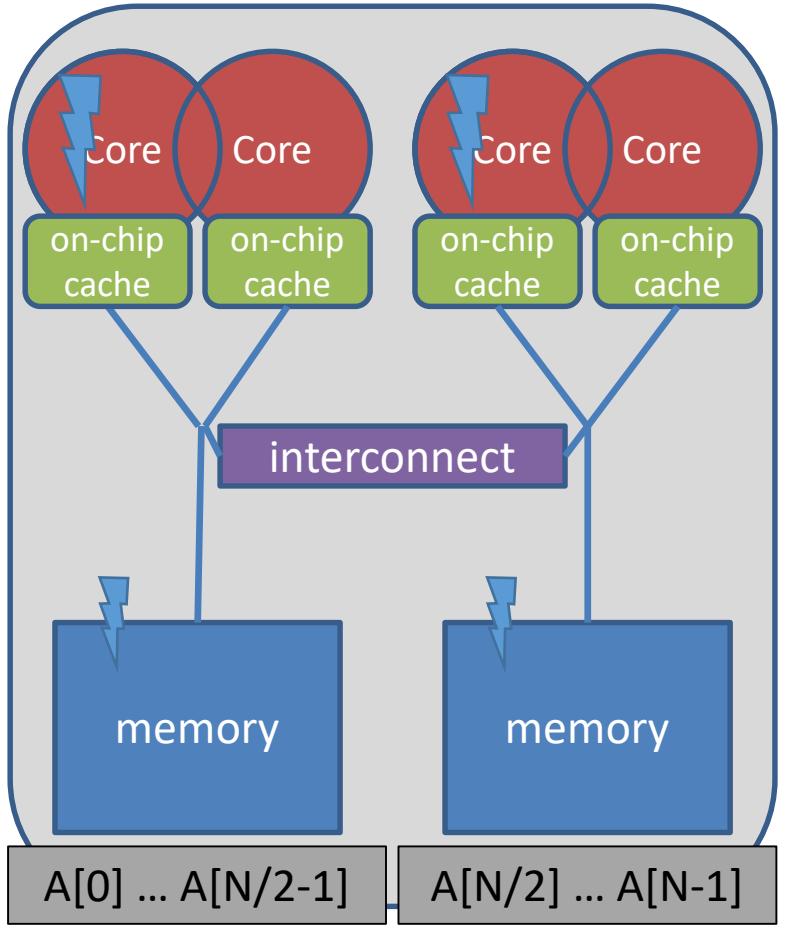
First Touch w/ parallel code: all array elements are allocated in the memory of the NUMA node that contains the core that executes the thread that initializes the partition

```
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;
}
```

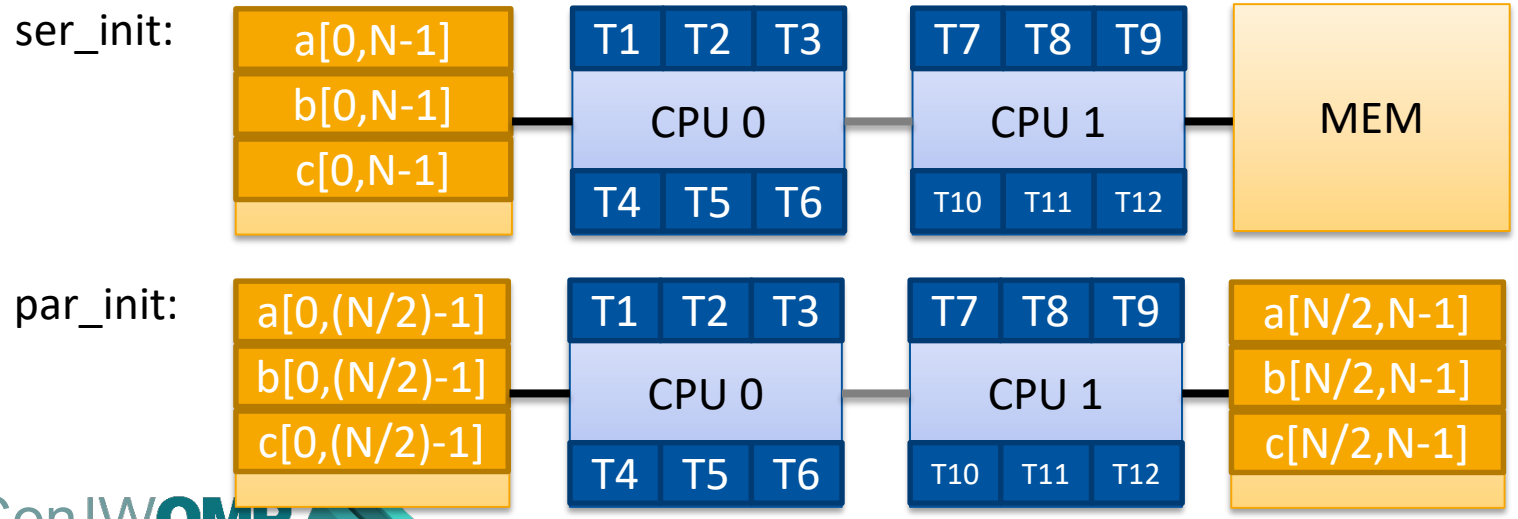
Static scheduling



Serial vs. Parallel Initialization

- Stream example with and without parallel initialization
  - 2 socket system 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



Get Information about the System Topology

- Before you design a strategy for thread binding, you should have a basic understanding of the system topology
- Options:
  - 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

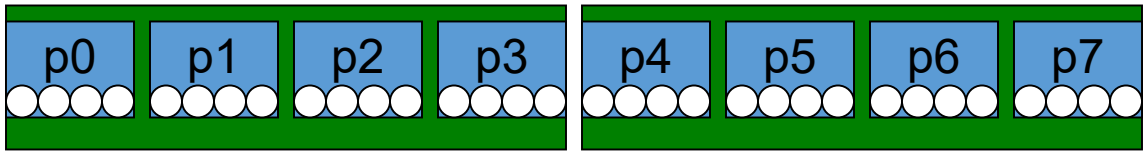
- Selecting the „right“ binding strategy depends not only on the topology, but also on the characteristics of your application
  - Putting threads far apart, i.e., on different sockets
    - May improve the aggregated memory bandwidth available to your application
    - May improve the combined cache 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** effective cache size
- If you are unsure, just try a few options and then select the best one

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 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

OMP_PLACES Environment Variable

- Assume the following machine:



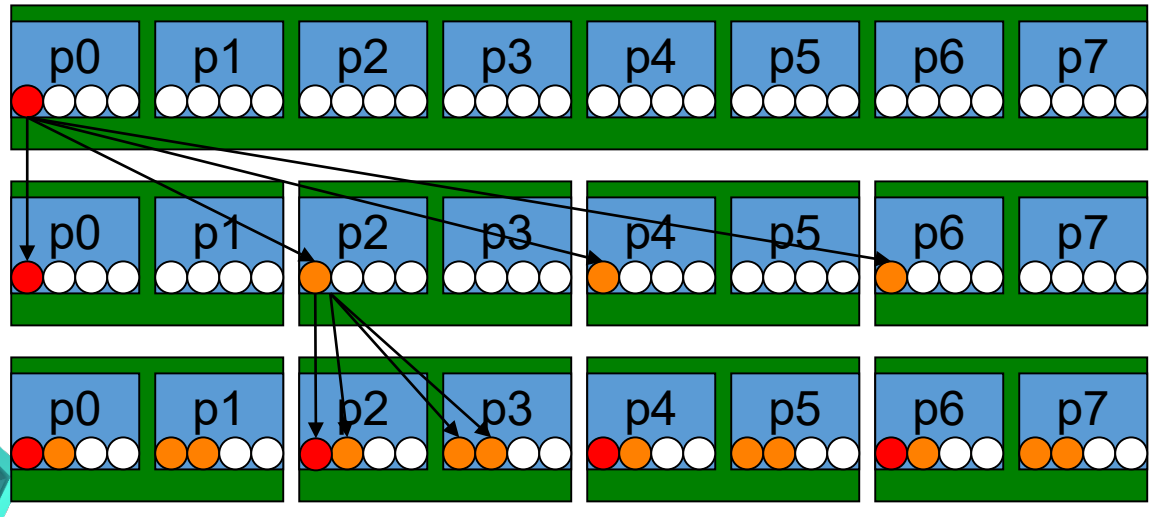
  - 2 sockets, 4 cores per socket, 4 hyper-threads per core

- Abstract names for OMP_PLACES:
  - threads: Each place corresponds to a single hardware thread 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.

OpenMP 4.0: Places and Binding Policies

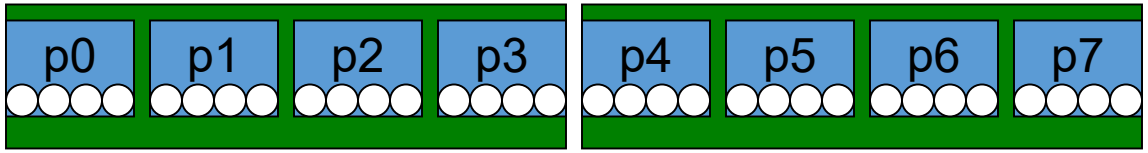
- Example‘s objective:
  - separate cores for outer loop and near cores for inner loop
- Outer parallel region: proc_bind(spread), Inner: proc_bind(close)
  - **spread** creates partition, **close** binds threads within respective partition

```
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)
```



More Examples (1/3)

- Assume the following machine:

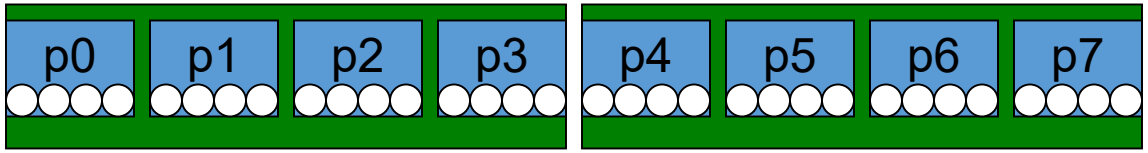


  - 2 sockets, 4 cores per socket, 4 hyper-threads per core

- 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:



- 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)
    ```
    ```

Looking at the execution: top

- Processor binding example
  - OMP_PROC_BIND=true ./matmul
    - Binds master thread to hw thread 0, thread 1 to hw thread 1...
  - OMP_PROC_BIND=close OMP_NUM_THREADS=2 ./matmul
    - Binds master thread to hw thread 0, thread 1 to hw thread 1

```
Cpu0  : 99.7%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.3%si,  0.0%st
Cpu1  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu2  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu3  :  0.0%us,  0.0%sy,  0.0%ni, 98.3%id,  1.7%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu4  : 98.0%us,  1.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  1.0%si,  0.0%st
Cpu5  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu6  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu7  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
```

1 socket, 4 cores per socket, 2 hyper-threads per core

Looking at the execution

- Processor binding example
  - OMP_PROC_BIND=spread OMP_NUM_THREADS=2 ./matmul
    - Binds master thread to hw thread 0, thread 1 to hw thread NUMPROCs/2

```
Cpu0  : 99.3%us,  0.0%sy,  0.0%ni,  0.7%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu1  :  0.0%us,  0.3%sy,  0.0%ni, 99.7%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu2  : 87.1%us,  0.0%sy,  0.0%ni, 12.9%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu3  :  0.0%us,  0.3%sy,  0.0%ni, 99.7%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu4  :  0.0%us,  1.0%sy,  0.0%ni, 66.0%id, 33.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu5  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu6  :  0.3%us,  0.0%sy,  0.0%ni, 99.7%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu7  :  0.3%us,  0.3%sy,  0.0%ni, 99.3%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
```


1 socket, 4 cores per socket, 2 hyper-threads per core

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);
}
```

Retrieving information from OpenMP

- OMP_DISPLAY_ENV=TRUE

```
OPENMP DISPLAY ENVIRONMENT BEGIN
  _OPENMP = '201511'
  OMP_DYNAMIC = 'FALSE'
  OMP_NESTED = 'FALSE'
  OMP_NUM_THREADS = ‘2'
  OMP_SCHEDULE = 'DYNAMIC'
  OMP_PROC_BIND = ‘SPREAD'
  OMP_PLACES = '{0},{4},{1},{5},{2},{6},{3},{7}'
  OMP_STACKSIZE = '0'
  OMP_WAIT_POLICY = 'PASSIVE'
  OMP_THREAD_LIMIT = '4294967295'
  OMP_MAX_ACTIVE_LEVELS = '2147483647'
  OMP_CANCELLATION = 'FALSE'
  OMP_DEFAULT_DEVICE = '0'
  OMP_MAX_TASK_PRIORITY = '0'
OPENMP DISPLAY ENVIRONMENT END
```

VERBOSE

```
OPENMP DISPLAY ENVIRONMENT BEGIN
  _OPENMP = '201511'
  OMP_DYNAMIC = 'FALSE'
  OMP_NESTED = 'FALSE'
  OMP_NUM_THREADS = '2'
  OMP_SCHEDULE = 'DYNAMIC'
  OMP_PROC_BIND = 'SPREAD'
  OMP_PLACES = '{0},{4},{1},{5},{2},{6},{3},{7}'
  OMP_STACKSIZE = '0'
  OMP_WAIT_POLICY = 'PASSIVE'
  OMP_THREAD_LIMIT = '4294967295'
  OMP_MAX_ACTIVE_LEVELS = '2147483647'
  OMP_CANCELLATION = 'FALSE'
  OMP_DEFAULT_DEVICE = '0'
  OMP_MAX_TASK_PRIORITY = '0'
  GOMP_CPU_AFFINITY = ''
  GOMP_STACKSIZE = '0'
  GOMP_SPINCOUNT = '300000'
OPENMP DISPLAY ENVIRONMENT END
```

A First Summary

- Everything is under control now?
- In principle yes, but only if
  - threads can be bound explicitly,
  - data can be placed well by first-touch, or can be migrated,
  - you focus on a specific platform (= os + arch) → no portability

- What if the data access pattern changes over time?

- What if you use more than one level of parallelism?

NUMA Strategies: Overview

- First Touch: Modern operating systems (i.e., Linux >= 2.4) determine the physical location of a memory page during the first page fault, when the page is first „touched“, and put it close to the CPU that causes the page fault
- Explicit Migration: Selected regions of memory (pages) are moved from one NUMA node to another via explicit OS syscall
- Next Touch: The binding of pages to NUMA nodes is removed and pages are put in the location of the next „touch“; well supported in Solaris, expensive to implement in Linux
- Automatic Migration: No support for this in current operating systems
  - Old SGI IRIX systems supported automatic migration

User 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`

OpenMP Memory Allocators (v5.0)

- New clause on all constructs with data sharing clauses:
  - `allocate( [allocator:] list )`
- Allocation:
  - `omp_alloc(size_t size, omp_allocator_t *allocator)`
- Deallocation:
  - `omp_free(void *ptr, const omp_allocator_t *allocator)`
  - `allocator` argument is optional
- `allocate` directive
  - Standalone directive for allocation, or declaration of allocation stmt.

Example: Using Memory Allocators (v5.0)

```
void allocator_example(omp_allocator_t *my_allocator) {
    int a[M], b[N], c;
    #pragma omp allocate(a) allocator(omp_high_bw_mem_alloc)
    #pragma omp allocate(b) // controlled by OMP_ALLOCATOR and/or omp_set_default_allocator
    double *p = (double *) omp_alloc(N*M*sizeof(*p), my_allocator);

    #pragma omp parallel private(a) allocate(my_allocator:a)
    {
        some_parallel_code();
    }

    #pragma omp target firstprivate(c) allocate(omp_const_mem_alloc:c)  // on target; must be compile-time expr
    {
         #pragma omp parallel private(a) allocate(omp_high_bw_mem_alloc:a)
         {
             some_other_parallel_code();
          }
    }

    omp_free(p);
}
```

OpenMP Task Affinity (v5.0)

- OpenMP version 5.0 will support task affinity

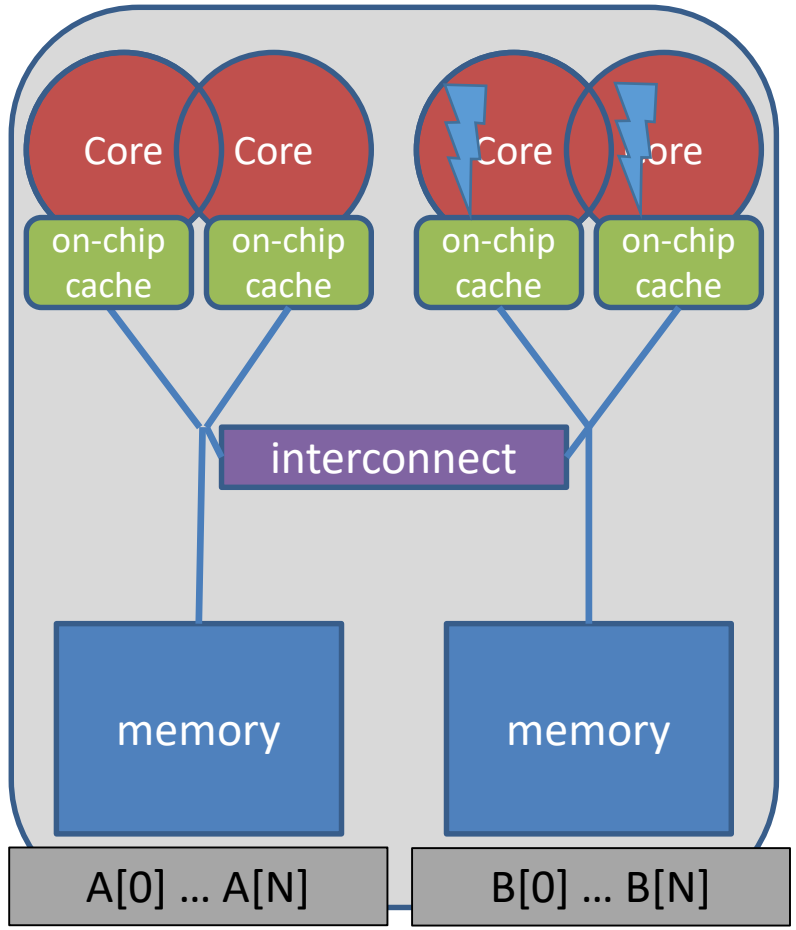
```
#pragma omp task affinity(<var-reference>)
```

  - Task-to-data affinity
  - Hint to execute task as close as possible to the location of the data

- Very similar information to target map(...) and task depend(...)

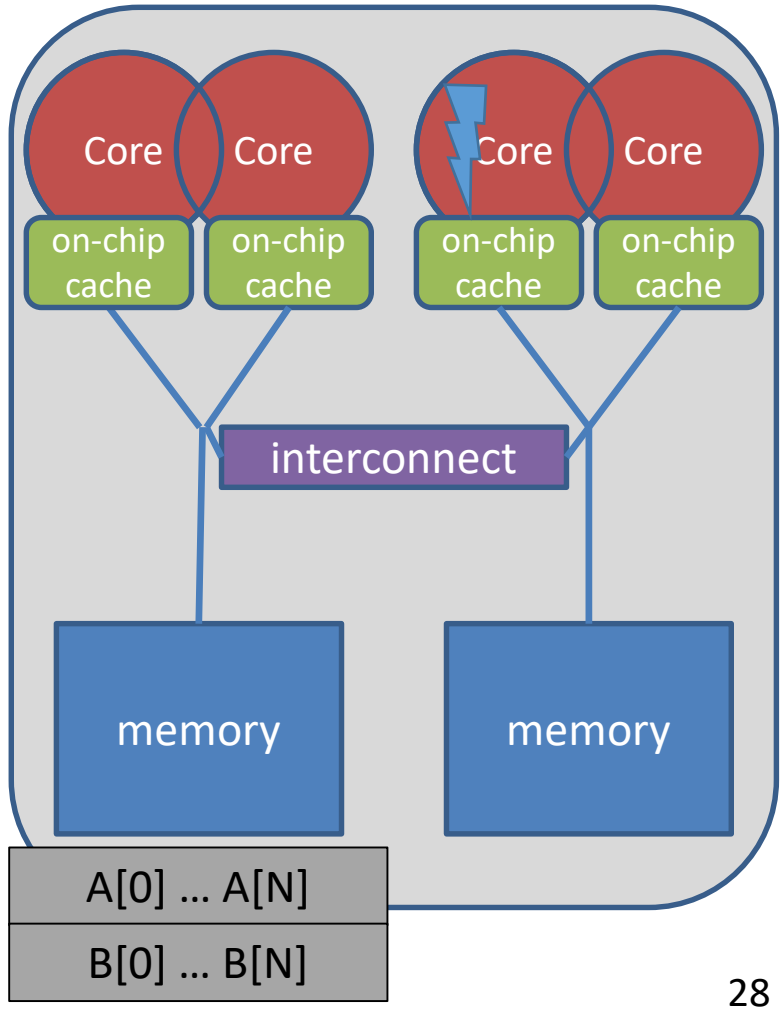
OpenMP Task Affinity

```
void task_affinity() {
    double* B;
#pragma omp task shared(B)
    {
        B = init_B_and_important_computation(A);
    }
#pragma omp task firstprivate(B)
    {
        important_computation_too(B);
    }
#pragma omp taskwait
}
```



OpenMP Task Affinity

```
void task_affinity() {
    double* B;
#pragma omp task shared(B) affinity(A[0:N])
    {
        B = init_B_and_important_computation(A);
    }
#pragma omp task firstprivate(B) affinity(B[0:N])
    {
        important_computation_too(B);
    }
#pragma omp taskwait
}
```



Partitioning Memory w/ OpenMP version 5.0

```
void allocator_example() {
    double *array;

    omp_allocator_t *allocator;
    omp_alloctrait_t traits[] = {
        {OMP_ATK_PARTITION, OMP_ATV_BLOCKED}
    };
    int ntraits = sizeof(traits) / sizeof(*traits);
    allocator = omp_init_allocator(omp_default_mem_space, ntraits, traits);

    array = omp_alloc(sizeof(*array) * N, allocator);

#pragma omp parallel for proc_bind(spread)
    for (int i = 0; i < N; ++i) {
        important_computation(&array[i]);
    }

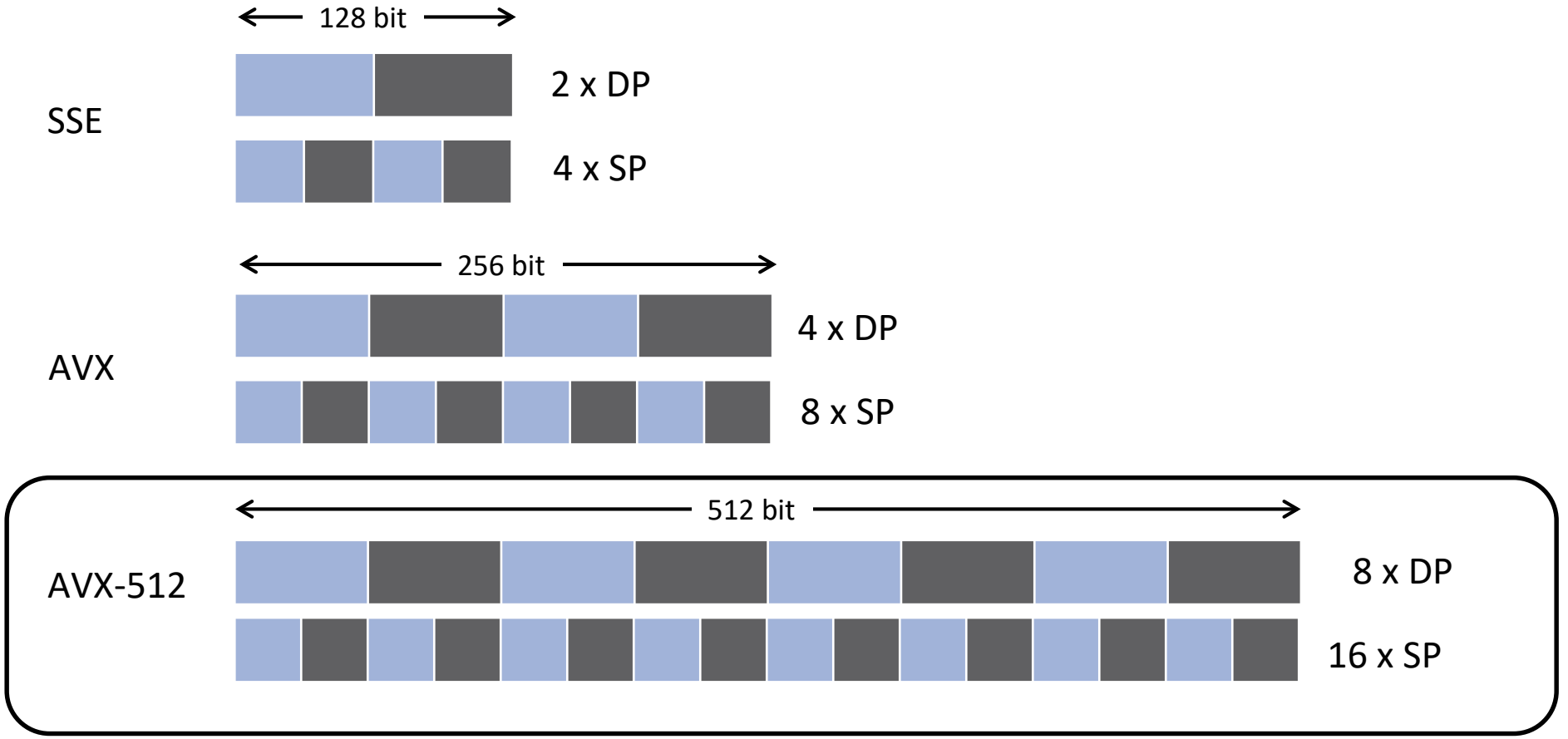
    omp_free(array);
}
```

Summary

- (Correct) memory placement is crucial for performance for most applications
- OpenMP programmers can exploit placement policies to align data with compute threads
- OpenMP version 5.0 will bring additional features for more portable memory optimizations

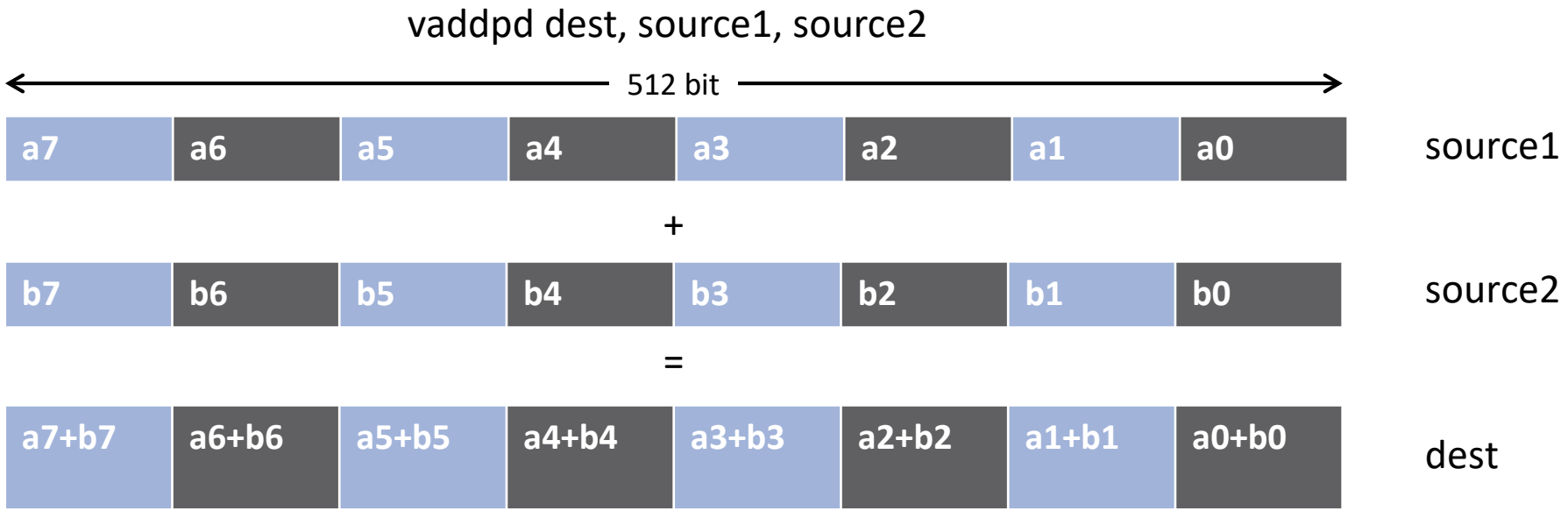
OpenMP SIMD Programming

Evolution of SIMD on Intel® Architectures



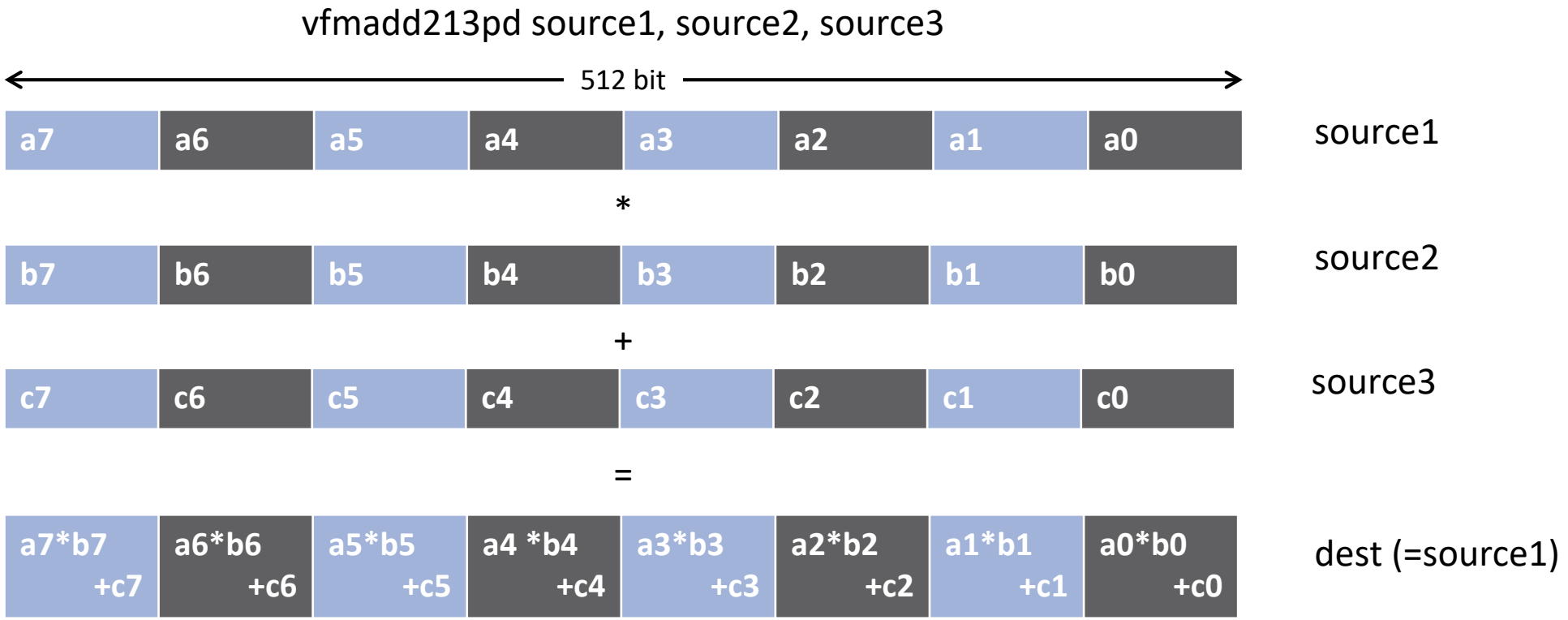
SIMD Instructions –Arithmetic Instructions

Operations work on each individual SIMD element



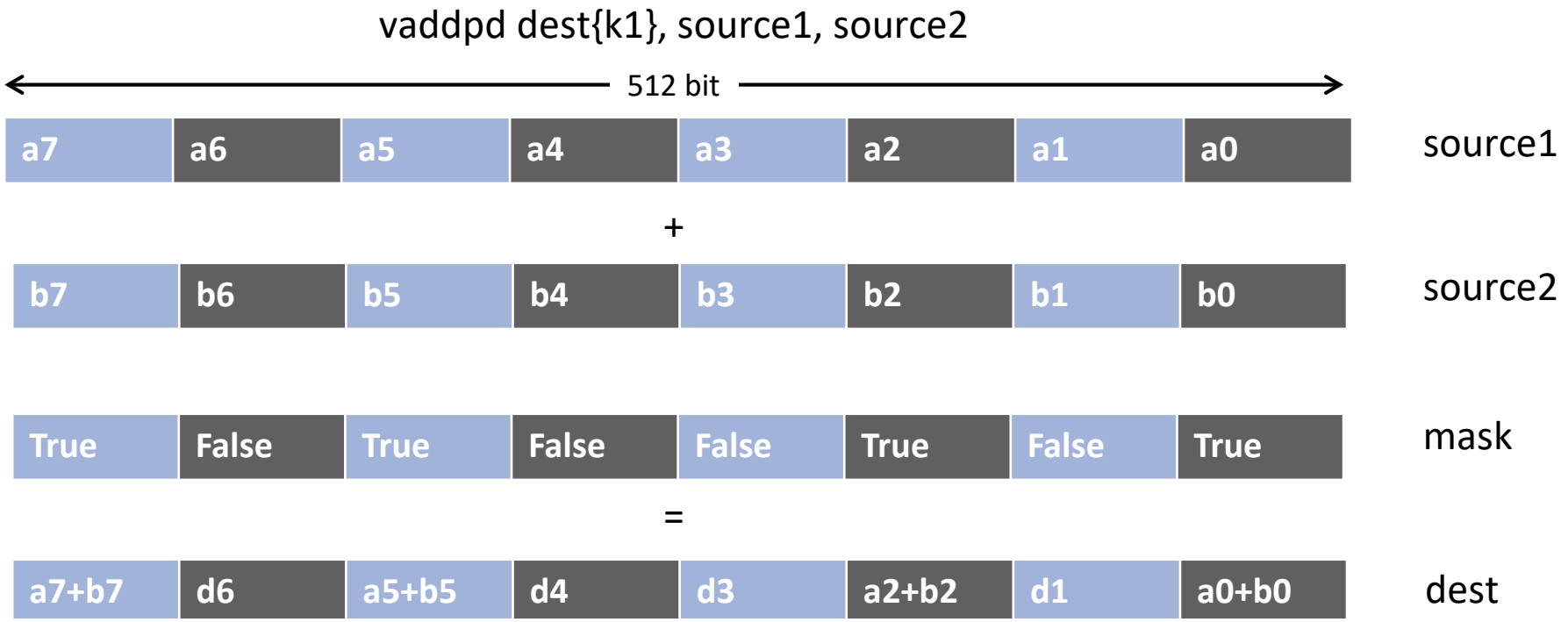
SIMD Instructions – Fused Instructions

Two operations (e.g., multiply & add) fused into one SIMD instruction



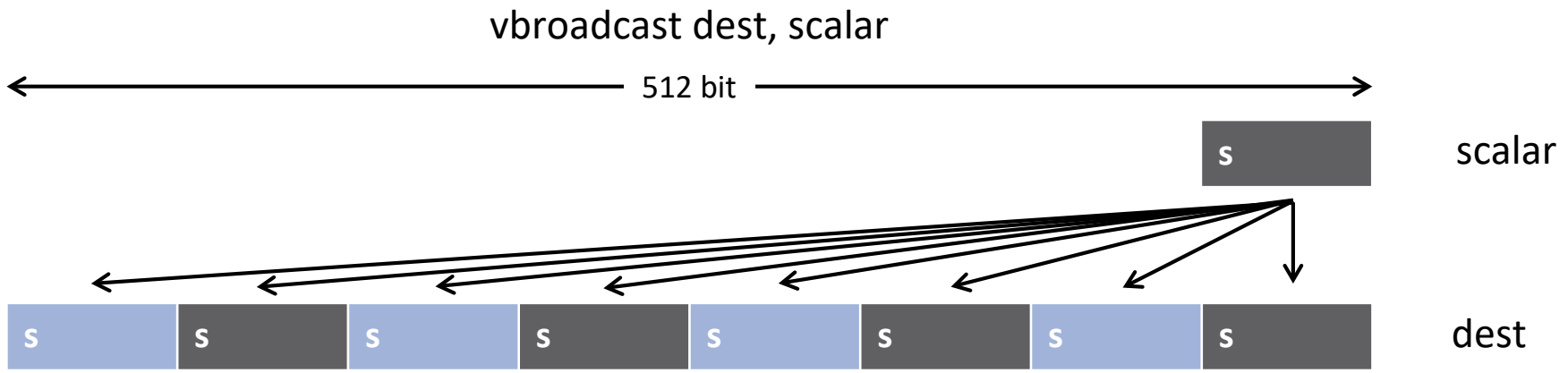
SIMD Instructions – Conditional Evaluation

Mask register limit effect of instructions to a subset of the SIMD elements



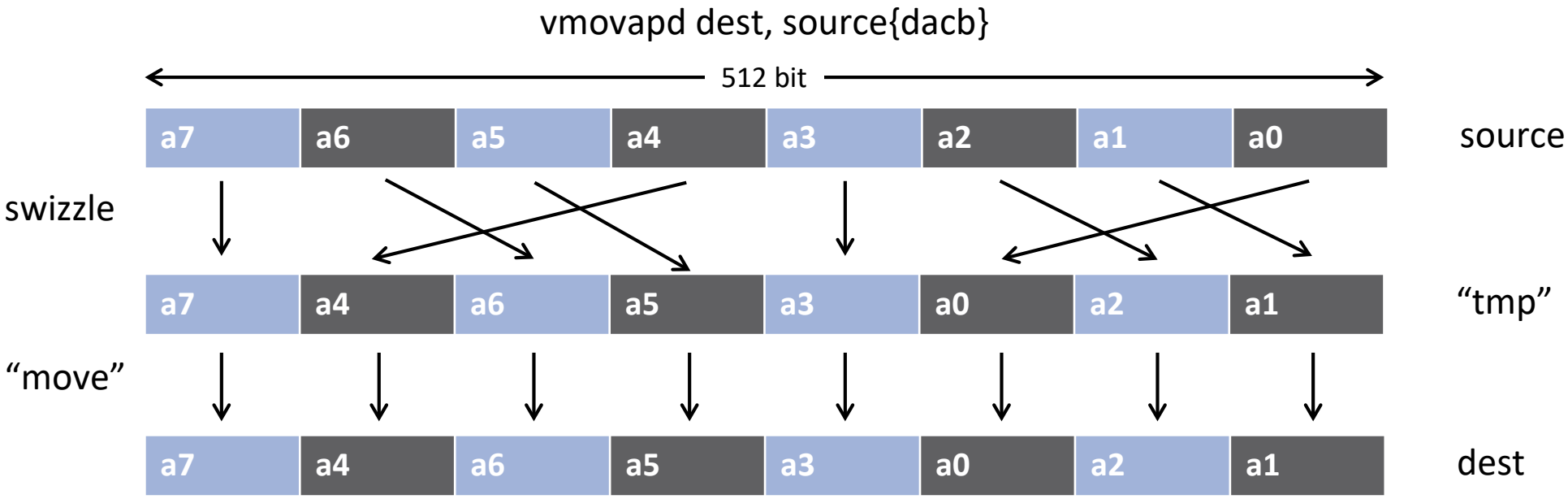
SIMD Instructions – Broadcast

Assign a scalar value to all SIMD elements



SIMD Instructions – Shuffles, Swizzles, Blends

Instruction to modify data layout in the SIMD register



Auto-vectorization

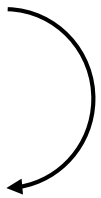
- Compilers offer auto-vectorization as an optimization pass
  - Usually part of the general loop optimization passes
  - Code analysis detects code properties that inhibit SIMD vectorization
  - Heuristics determine if SIMD execution might be beneficial
  - If all goes well, the compiler will generate SIMD instructions
- Example: Intel® Composer XE
  - -vec (automatically enabled with –O2)
  - -qopt-report

Interlude: 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
```

Interlude: Loop-carried Dependencies

- Dependencies may occur across loop iterations
  - Then they are called “loop-carried dependencies”
  - “Distance” of a dependency: number of loop iterations the dependency spans
- The following code contains such a dependency:

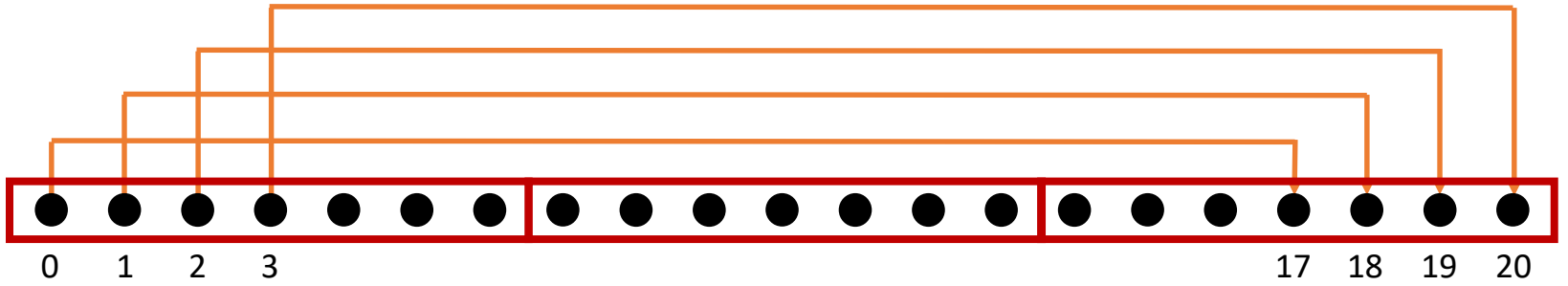
```
void lcd_ex(float* a, float* b, size_t n, float c1, float c2) {
    for (int i = 0; i < n; i++) {
        a[i] = c1 * a[i + 17] + c2 * b[i];
}   }
```

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?
  - Note: This condition is sufficient, but not necessary!

Interlude: 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)

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

Example: Loop not Countable

- “Loop not Countable” plus “Assumed Dependencies”

```
typedef struct {
    float* data;
    int size;
} vec_t;

void vec_eltwise_product(vec_t* a, vec_t* b, vec_t* c) {
    for (int i = 0; i < a->size; i++) {
        c->data[i] = a->data[i] * b->data[i];
    }
}
```

OpenMP 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++)

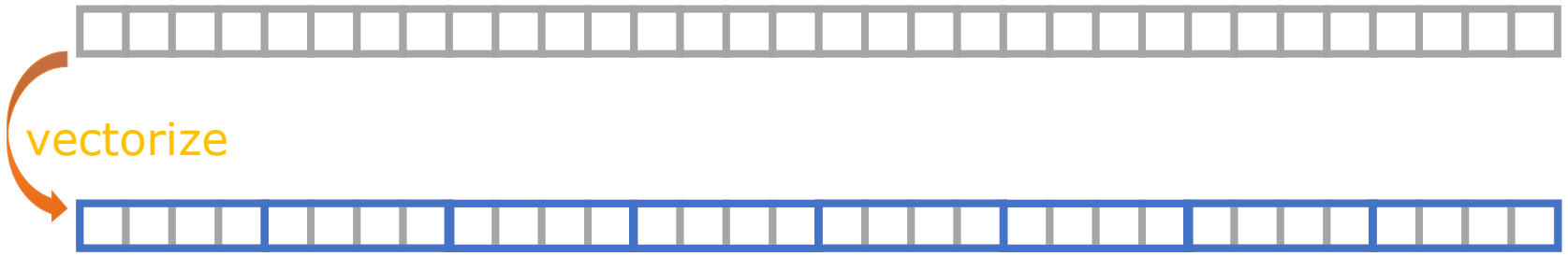
```
#pragma omp simd [clause[[,] clause],…]
for-loops
```

- Syntax (Fortran)

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

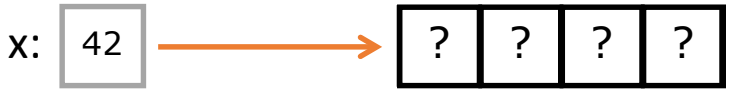
Example

```
void 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;
}
```

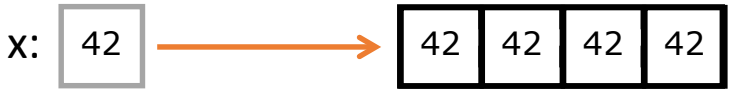


Data Sharing Clauses

- `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++)

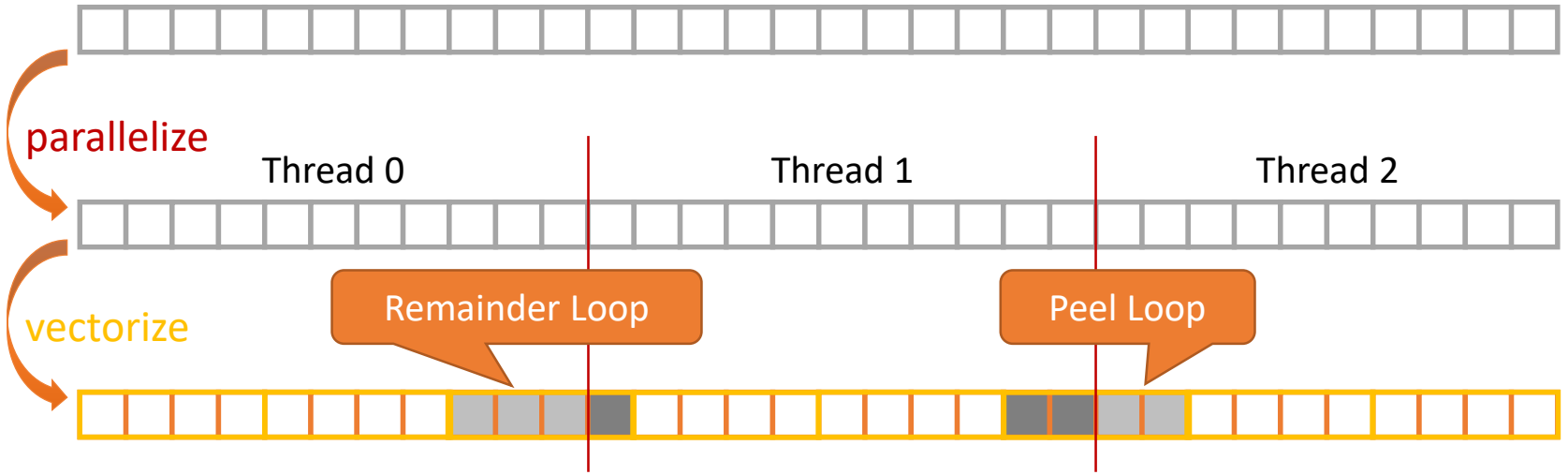
```
#pragma omp for simd [clause[[,] clause],…]
for-loops
```

- Syntax (Fortran)

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

Example

```
void 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…

```
void 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 (= 8-wide), the code will only execute the remainder loop!
  - and SSE (=4-wide), the code will have one iteration in the SIMD loop plus one in the remainder loop!

OpenMP 4.5 SIMD Chunks

```
void 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]
}   }
```

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

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

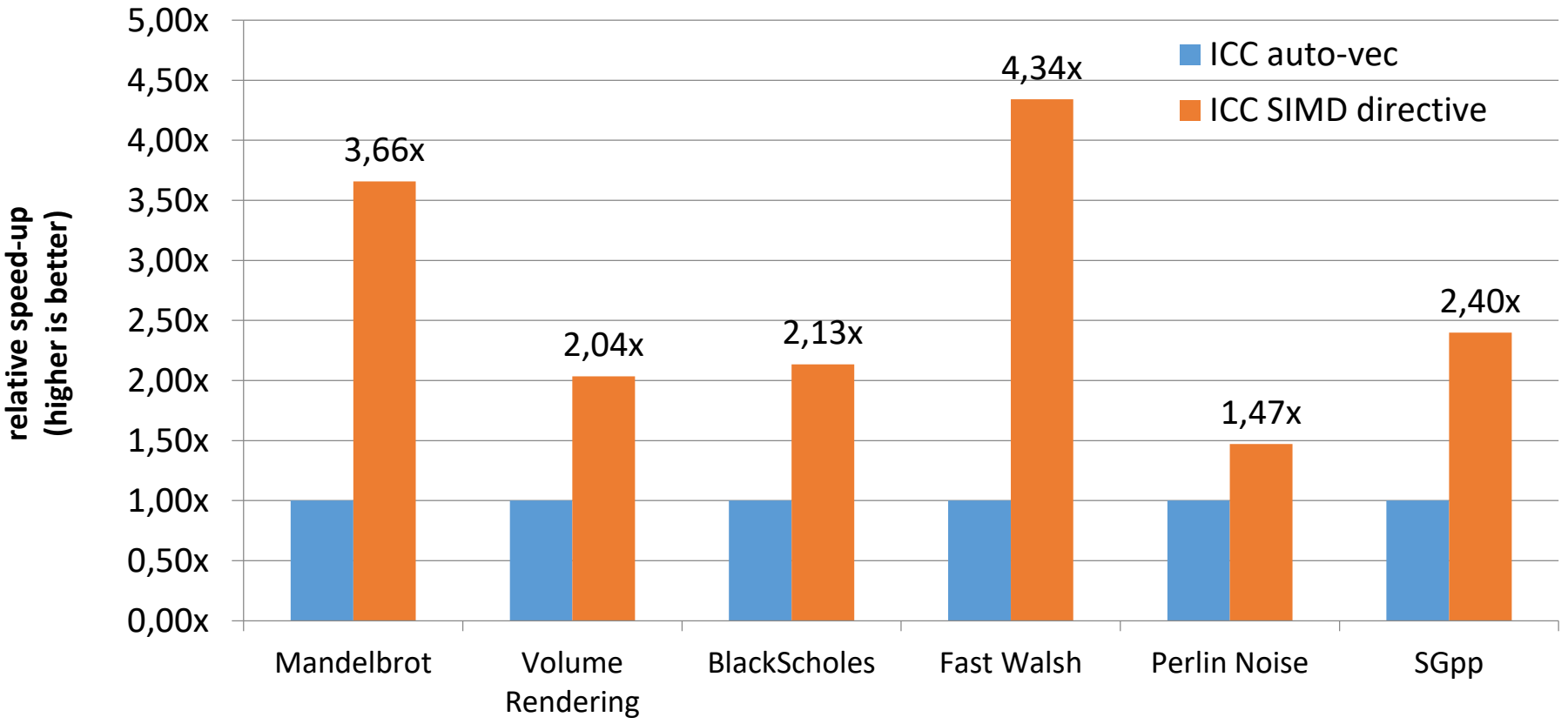
```
vmovups (%r14,%r12,4), %zmm0
vmovups (%r13,%r12,4), %zmm1
call _ZGVZN16vv_distsq
vmovups (%rbx,%r12,4), %zmm1
call _ZGVZN16vv_min
```

AT&T syntax: destination operand is on the right

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`
  - optimize for 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]`*`)`

SIMD Constructs & Performance



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.

Visit www.openmp.org

Summary

- OpenMP provided a powerful, expressive tasking model
- NUMA-aware programming is essential for performance
- OpenMP supports data-parallel instructions through the semi-automatic SIMD features
- Connect with us to share feedback, comments, concerns, propose features, or just hang around and have fun

Updated Slides

http://bit.ly/201809tutorial