2DV605

Parallel Computing

OpenMP

Sabri Pllana, PhD

Department of Computer Science

Building D, Room D2236C, LNU, Växjö

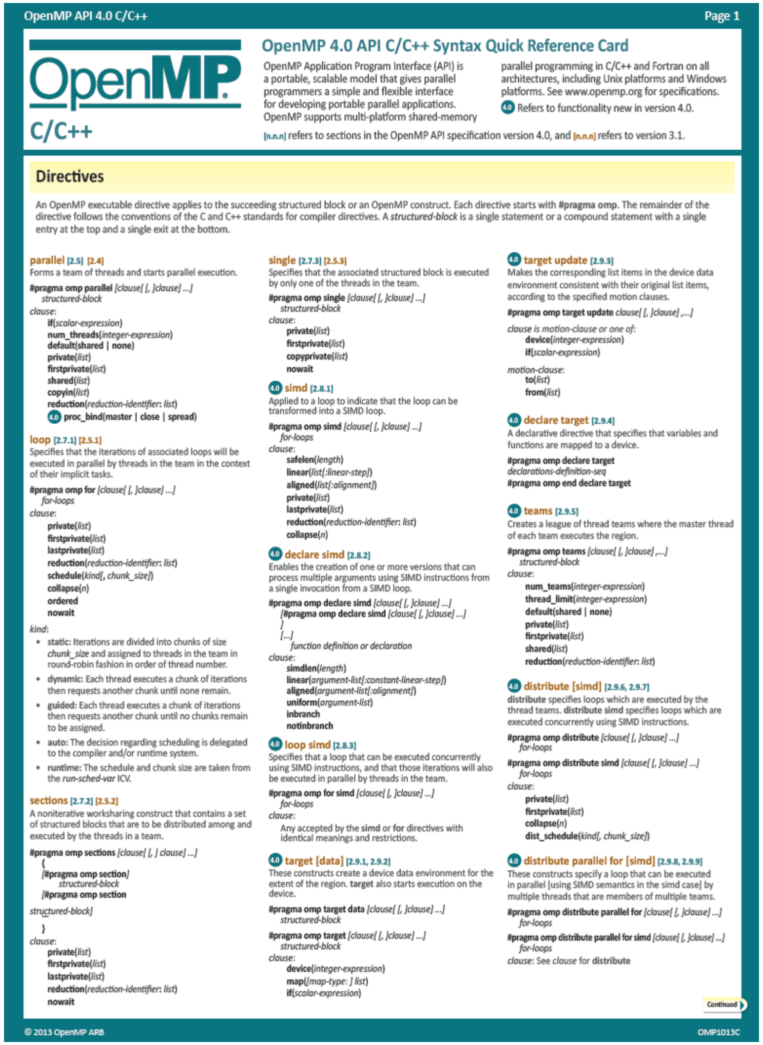
sabri.pllana@lnu.se, http://homepage.lnu.se/staff/saplaa/

Outline

- ❑ Introduction
- ❑ Example: Hello World
- ❑ Compiler Directives
- ❑ Runtime Library Routines
- ❑ Environment Variables
- ❑ Example: Matrix Multiplication

Selected Literature

- ❑ **OpenMP Specifications**
  - Version 4.0, July 2013 (supported by GCC 4.9 or later version)
  - OpenMP 4.5, November 2015
  - http://openmp.org/wp/openmp-specifications/
- ❑ **OpenMP 4.0 API C/C++ Syntax Quick Reference Card**
  - http://openmp.org/mp-documents/OpenMP-4.0-C.pdf



OpenMP API 4.0 C/C++ Page 1

OpenMP C/C++

OpenMP 4.0 API C/C++ Syntax Quick Reference Card

OpenMP Application Program Interface (API) is a portable, scalable model that gives parallel programmers a simple and flexible interface for developing portable parallel applications. OpenMP supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures, including Unix platforms and Windows platforms. See www.openmp.org for specifications.

4.0 Refers to functionality new in version 4.0.

[n.n.n] refers to sections in the OpenMP API specification version 4.0, and [n.n.n] refers to version 3.1.

Directives

An OpenMP executable directive applies to the succeeding structured block or an OpenMP construct. Each directive starts with **#pragma omp**. The remainder of the directive follows the conventions of the C and C++ standards for compiler directives. A *structured-block* is a single statement or a compound statement with a single entry at the top and a single exit at the bottom.

parallel [2.5] [2.4]

Forms a team of threads and starts parallel execution.

```
#pragma omp parallel [clause[ [, ]clause] ...]
    structured-block
clause:
    if(scalar-expression)
    num_threads(integer-expression)
    default(shared | none)
    private(list)
    firstprivate(list)
    shared(list)
    copyin(list)
    reduction(reduction-identifier: list)
    4.0 proc_bind(master | close | spread)
```

loop [2.7.1] [2.5.1]

Specifies that the iterations of associated loops will be executed in parallel by threads in the team in the context of their implicit tasks.

```
#pragma omp for [clause[ [, ]clause] ...]
    for-loops
clause:
    private(list)
    firstprivate(list)
    lastprivate(list)
    reduction(reduction-identifier: list)
    schedule(kind[, chunk_size])
    collapse(n)
    ordered
    nowait
```

kind:

- **static**: Iterations are divided into chunks of size *chunk_size* and assigned to threads in the team in round-robin fashion in order of thread number.
- **dynamic**: Each thread executes a chunk of iterations then requests another chunk until none remain.
- **guided**: Each thread executes a chunk of iterations then requests another chunk until no chunks remain to be assigned.
- **auto**: The decision regarding scheduling is delegated to the compiler and/or runtime system.
- **runtime**: The schedule and chunk size are taken from the *run-sched-var* ICV.

sections [2.7.2] [2.5.2]

A noniterative worksharing construct that contains a set of structured blocks that are to be distributed among and executed by the threads in a team.

```
#pragma omp sections [clause[ [, ] clause] ...]
    {
    [#pragma omp section]
        structured-block
    [#pragma omp section
structured-block]
    ...
    }
clause:
    private(list)
    firstprivate(list)
    lastprivate(list)
    reduction(reduction-identifier: list)
    nowait
```

single [2.7.3] [2.5.3]

Specifies that the associated structured block is executed by only one of the threads in the team.

```
#pragma omp single [clause[ [, ]clause] ...]
    structured-block
clause:
    private(list)
    firstprivate(list)
    copyprivate(list)
    nowait
```

4.0 **simd** [2.8.1]

Applied to a loop to indicate that the loop can be transformed into a SIMD loop.

```
#pragma omp simd [clause[ [, ]clause] ...]
    for-loops
clause:
    safelen(length)
    linear(list[:linear-step])
    aligned(list[:alignment])
    private(list)
    lastprivate(list)
    reduction(reduction-identifier: list)
    collapse(n)
```

4.0 **declare simd** [2.8.2]

Enables the creation of one or more versions that can process multiple arguments using SIMD instructions from a single invocation from a SIMD loop.

```
#pragma omp declare simd [clause[ [, ]clause] ...]
    [#pragma omp declare simd [clause[ [, ]clause] ...]
    ]
    [...]
      function definition or declaration
clause:
    simdlen(length)
    linear(argument-list[:constant-linear-step])
    aligned(argument-list[:alignment])
    uniform(argument-list)
    inbranch
    notinbranch
```

4.0 **loop simd** [2.8.3]

Specifies that a loop that can be executed concurrently using SIMD instructions, and that those iterations will also be executed in parallel by threads in the team.

```
#pragma omp for simd [clause[ [, ]clause] ...]
    for-loops
clause:
   Any accepted by the simd or for directives with
   identical meanings and restrictions.
```

4.0 **target [data]** [2.9.1, 2.9.2]

These constructs create a device data environment for the extent of the region. **target** also starts execution on the device.

```
#pragma omp target data [clause[ [, ]clause] ...]
    structured-block

#pragma omp target [clause[ [, ]clause] ...]
    structured-block
clause:
    device(integer-expression)
    map([map-type: ] list)
    if(scalar-expression)
```

4.0 **target update** [2.9.3]

Makes the corresponding list items in the device data environment consistent with their original list items, according to the specified motion clauses.

```
#pragma omp target update clause[ [, ]clause] ,...]

clause is motion-clause or one of:
    device(integer-expression)
    if(scalar-expression)

motion-clause:
    to(list)
    from(list)
```

4.0 **declare target** [2.9.4]

A declarative directive that specifies that variables and functions are mapped to a device.

```
#pragma omp declare target
declarations-definition-seq
#pragma omp end declare target
```

4.0 **teams** [2.9.5]

Creates a league of thread teams where the master thread of each team executes the region.

```
#pragma omp teams [clause[ [, ]clause] ,...]
    structured-block
clause:
    num_teams(integer-expression)
    thread_limit(integer-expression)
    default(shared | none)
    private(list)
    firstprivate(list)
    shared(list)
    reduction(reduction-identifier: list)
```

4.0 **distribute [simd]** [2.9.6, 2.9.7]

distribute specifies loops which are executed by the thread teams. **distribute simd** specifies loops which are executed concurrently using SIMD instructions.

```
#pragma omp distribute [clause[ [, ]clause] ...]
    for-loops

#pragma omp distribute simd [clause[ [, ]clause] ...]
    for-loops
clause:
    private(list)
    firstprivate(list)
    collapse(n)
    dist_schedule(kind[, chunk_size])
```

4.0 **distribute parallel for [simd]** [2.9.8, 2.9.9]

These constructs specify a loop that can be executed in parallel [using SIMD semantics in the simd case] by multiple threads that are members of multiple teams.

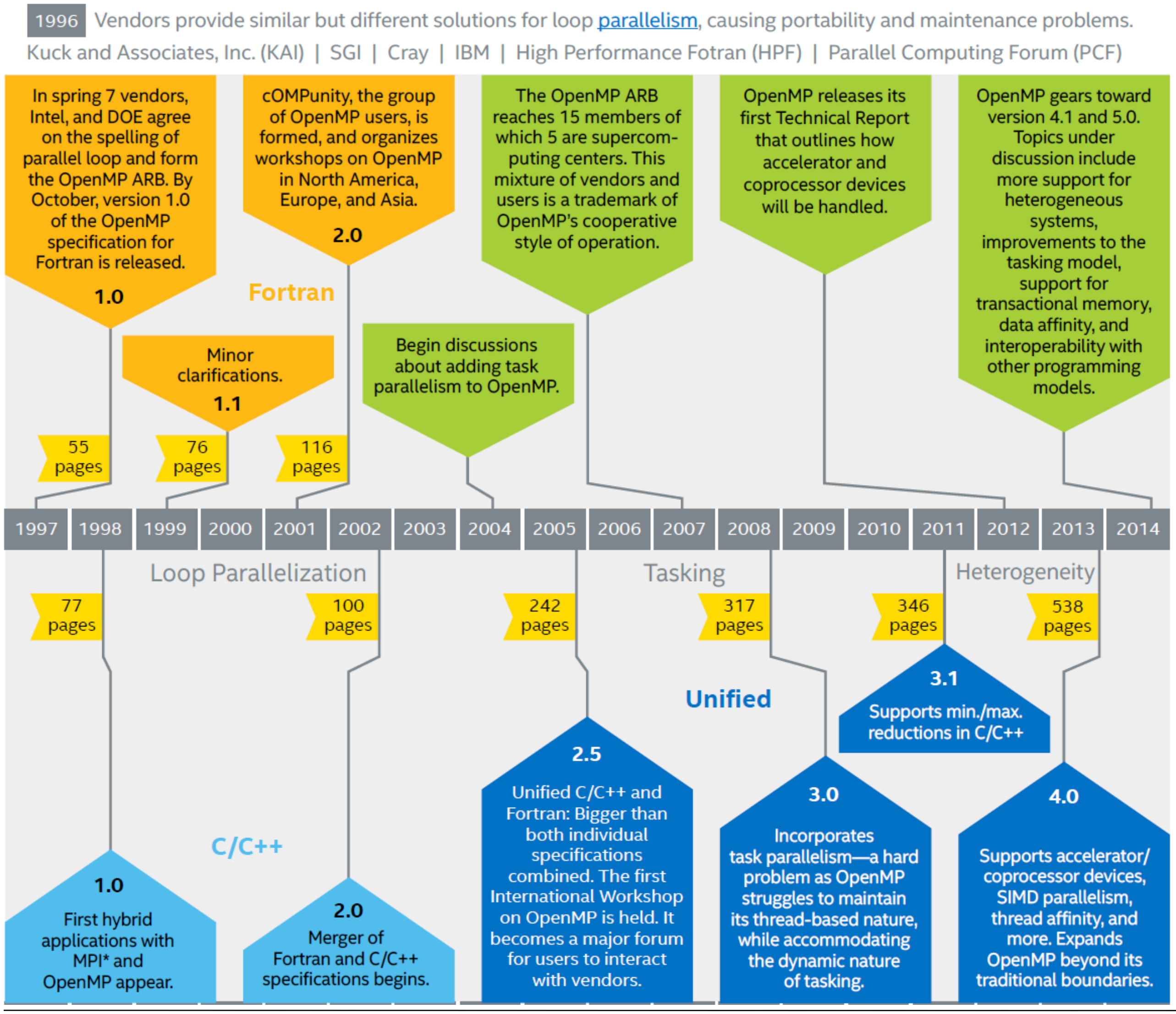
```
#pragma omp distribute parallel for [clause[ [, ]clause] ...]
    for-loops

#pragma omp distribute parallel for simd [clause[ [, ]clause] ...]
    for-loops
clause: See clause for distribute
```

Continued

© 2013 OpenMP ARB OMP1013C

Introduction

OpenMP History



1996 Vendors provide similar but different solutions for loop parallelism, causing portability and maintenance problems.

Kuck and Associates, Inc. (KAI) | SGI | Cray | IBM | High Performance Fotran (HPF) | Parallel Computing Forum (PCF)

In spring 7 vendors, Intel, and DOE agree on the spelling of parallel loop and form the OpenMP ARB. By October, version 1.0 of the OpenMP specification for Fortran is released. 1.0

cOMPunity, the group of OpenMP users, is formed, and organizes workshops on OpenMP in North America, Europe, and Asia. 2.0

Fortran

Minor clarifications. 1.1

The OpenMP ARB reaches 15 members of which 5 are supercomputing centers. This mixture of vendors and users is a trademark of OpenMP's cooperative style of operation.

Begin discussions about adding task parallelism to OpenMP.

OpenMP releases its first Technical Report that outlines how accelerator and coprocessor devices will be handled.

OpenMP gears toward version 4.1 and 5.0. Topics under discussion include more support for heterogeneous systems, improvements to the tasking model, support for transactional memory, data affinity, and interoperability with other programming models.

55 pages | 76 pages | 116 pages

1997 | 1998 | 1999 | 2000 | 2001 | 2002 | 2003 | 2004 | 2005 | 2006 | 2007 | 2008 | 2009 | 2010 | 2011 | 2012 | 2013 | 2014

Loop Parallelization | Tasking | Heterogeneity

77 pages | 100 pages | 242 pages | 317 pages | 346 pages | 538 pages

Unified

3.1 Supports min./max. reductions in C/C++

C/C++

1.0 First hybrid applications with MPI* and OpenMP appear.

2.0 Merger of Fortran and C/C++ specifications begins.

2.5 Unified C/C++ and Fortran: Bigger than both individual specifications combined. The first International Workshop on OpenMP is held. It becomes a major forum for users to interact with vendors.

3.0 Incorporates task parallelism—a hard problem as OpenMP struggles to maintain its thread-based nature, while accommodating the dynamic nature of tasking.

4.0 Supports accelerator/ coprocessor devices, SIMD parallelism, thread affinity, and more. Expands OpenMP beyond its traditional boundaries.

Credit: The Parallel Universe, June 2014, © Intel Corporation

OpenMP Architecture Review Board (ARB)

ARB oversees OpenMP specification

Permanent ARB members develop and sell OpenMP products

- AMD (Greg Stoner)
- ARM (Chris Adeniyi-Jones)
- Cray (Luiz DeRose)
- Fujitsu (Eiji Yamanaka)
- HP (Sujoy Saraswati)
- IBM (Kelvin Li)
- Intel (Xinmin Tian)
- Micron (Kirby Collins)
- NEC (Kazuhiro Kusano)
- NVIDIA (Jeff Larkin)
- Oracle Corporation (Nawal Copty)
- Red Hat (Matt Newsome)
- Texas Instruments (Eric Stotzer)

Auxiliary ARB members do not sell OpenMP products

- Argonne National Laboratory (Kalyan Kumaran)
- ASC/Lawrence Livermore National Laboratory (B. R. de Supinski)
- Barcelona Supercomputing Center (Xavier Martorell)
- Bristol University (Simon McIntosh-Smith)
- cOMPunity (Barbara Chapman/Yonghong Yan)
- Edinburgh Parallel Computing Centre (EPCC) (Mark Bull)
- INRIA (Olivier Aumage)
- Los Alamos National Laboratory (David Montoya)
- Lawrence Berkeley National Laboratory (Alice Koniges/Helen He)
- NASA (Henry Jin)
- Oak Ridge National Laboratory (Oscar Hernandez)
- RWTH Aachen University (Dieter an Mey)
- Sandia National Laboratory (Stephen Olivier)
- Texas Advanced Computing Center (Kent Milfeld)
- University of Houston (Deepak Eachempati/Jeremy Kemp)

Status: September 2016

OpenMP Overview

- ❑ **OpenMP supports shared-memory parallelism**
  - extends C, C++, Fortran programming languages

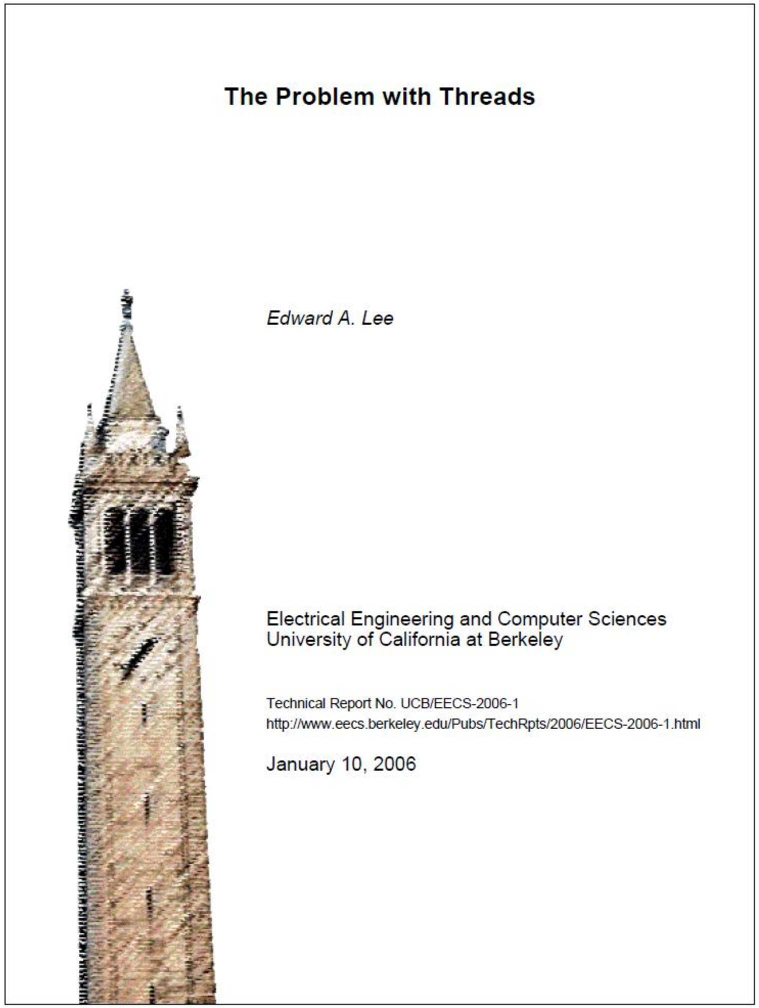
- ❑ **OpenMP provides constructs for**
  - single program multiple data (SPMD)
  - tasks
  - device
  - worksharing and synchronization

- ❑ **OpenMP (Application Programming Interface) API**
  - compiler directives
  - library routines
  - environment variables

Threads

- **Edward A. Lee (2006): The Problem with Threads**
  - *“Threads, as a model of computation, are wildly nondeterministic, and the job of the programmer becomes one of pruning that nondeterminism”*
  - *“non-trivial multi-threaded programs are incomprehensible to humans”*
  - *“Threads must be relegated to the engine room of computing, to be suffered only by expert technology providers”*

non determinism: various outputs may result for the same input



The Problem with Threads

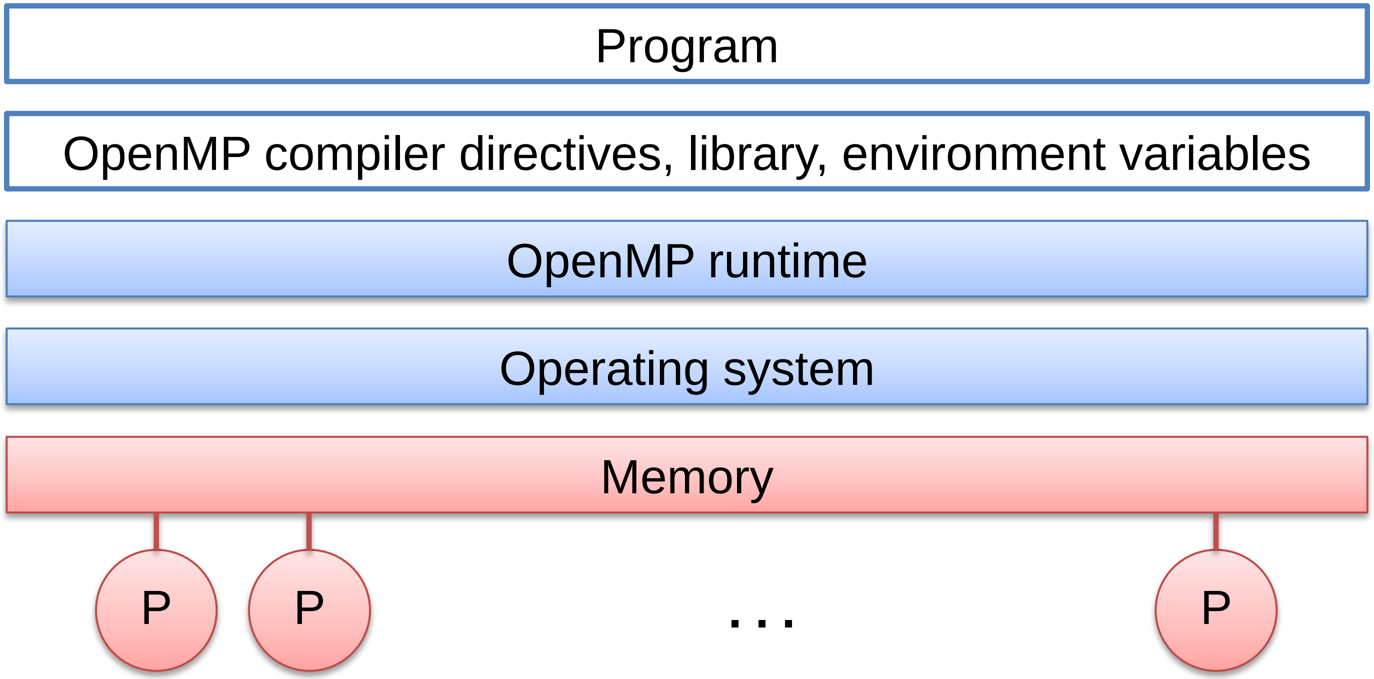
Edward A. Lee

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-1
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.html

January 10, 2006

OpenMP System Stack



Program

OpenMP compiler directives, library, environment variables

OpenMP runtime

Operating system

Memory

P P . . . P

OpenMP and Threads

```
#pragma omp parallel num_threads(4)
{
    foobar ();
}
```

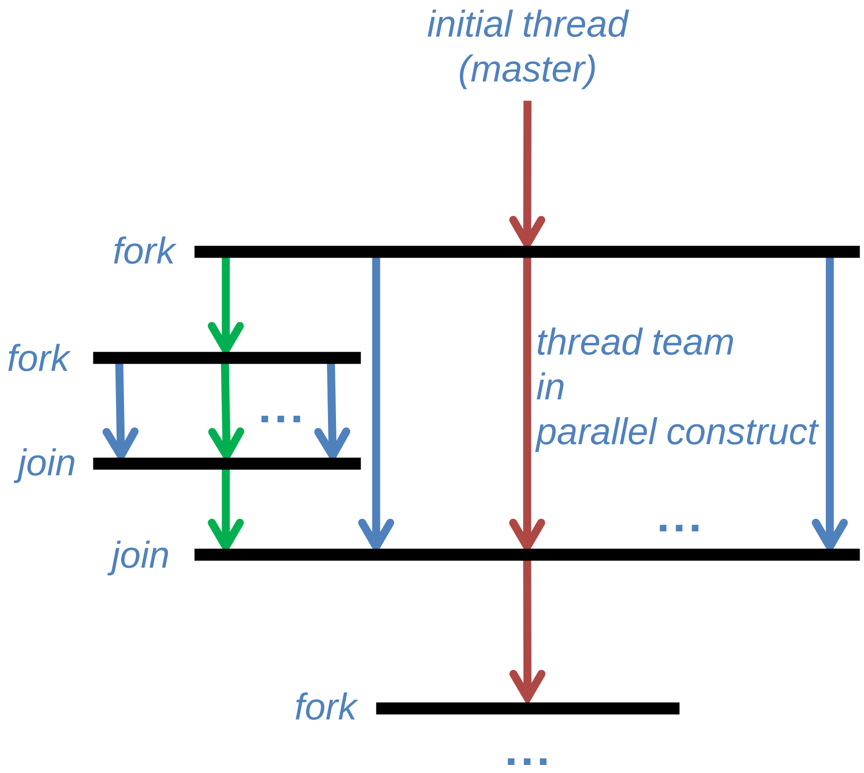
```
void thunk ()
{
    foobar ();
}

pthread_t tid[4];
for (int i = 1; i < 4; ++i)
   pthread_create (
         &tid[i],0,thunk, 0);
thunk();

for (int i = 1; i < 4; ++i)
    pthread_join (tid[i]);
```

Credit: Tim Mattson

OpenMP Execution Model

- ❑ **Fork and join model**
  - parallel regions may be nested
  - implicit barrier at the end of parallel construct
  - a program may comprise an arbitrary number of parallel constructs

- ❑ **No guarantee for synchronous file I/O during parallel execution**
  - programmer is responsible for synchronization, if multiple threads access the same file



Example: Hello World

Hello World: C-Code Extended with OpenMP

```
#include <omp.h>
#include <stdio.h>
main () {
int threads, id;
         #pragma omp parallel private(id)
         { // Begin of parallel region
         id = omp_get_thread_num(); // Get the thread ID
         printf("Hello World, I am thread = %d\n", id);
         // The thread with ID = 0 outputs the total number of threads
         if (id == 0) {
                  threads = omp_get_num_threads();
                  printf("Number of threads = %d\n", threads);
                  }
         } // End of parallel region
}
```

Hello World: Compilation and Execution

Compilation

```
-bash-4.1$ gcc -o hello -fopenmp hello.c
```

Execution on “Emil”

```
-bash-4.1$ ./hello
```

“Emil” comprises two processors of the type Intel Xeon E5-2695v2, each has 12 cores, and supports two threads per core (two logical cores per physical core).

In total are 48 threads.

```
Hello World, I am thread = 10
Hello World, I am thread = 12
...
Hello World, I am thread = 29
Hello World, I am thread = 15
Hello World, I am thread = 33
Hello World, I am thread = 28
Hello World, I am thread = 42
Hello World, I am thread = 8
Hello World, I am thread = 17
Hello World, I am thread = 18
Hello World, I am thread = 14
Hello World, I am thread = 47
Hello World, I am thread = 41
Hello World, I am thread = 44
Hello World, I am thread = 38
Hello World, I am thread = 37
Hello World, I am thread = 0
Number of threads = 48
Hello World, I am thread = 16
Hello World, I am thread = 22
Hello World, I am thread = 2
Hello World, I am thread = 1
Hello World, I am thread = 6
Hello World, I am thread = 26
```

Emil: CPU Information

```
-bash-4.1$ cat /proc/cpuinfo | more
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 62
model name      : Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz
stepping        : 4
cpu MHz         : 2399.828
cache size      : 30720 KB
…
processor       : 47
vendor_id       : GenuineIntel
cpu family      : 6
model           : 62
model name      : Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz
stepping        : 4
cpu MHz         : 2399.828
cache size      : 30720 KB
…
```

Environment Variable OMP_NUM_THREADS

Set the number of OpenMP threads to 12

```
-bash-4.1$ export OMP_NUM_THREADS=12
```

```
-bash-4.1$ ./hello
Hello World, I am thread = 10
Hello World, I am thread = 5
Hello World, I am thread = 2
Hello World, I am thread = 7
Hello World, I am thread = 6
Hello World, I am thread = 3
Hello World, I am thread = 4
Hello World, I am thread = 0
Number of threads = 12
Hello World, I am thread = 9
Hello World, I am thread = 11
Hello World, I am thread = 1
Hello World, I am thread = 8
```

```
-bash-4.1$ ./hello
Hello World, I am thread = 6
Hello World, I am thread = 11
Hello World, I am thread = 2
Hello World, I am thread = 4
Hello World, I am thread = 7
Hello World, I am thread = 5
Hello World, I am thread = 9
Hello World, I am thread = 10
Hello World, I am thread = 3
Hello World, I am thread = 8
Hello World, I am thread = 1
Hello World, I am thread = 0
Number of threads = 12
```

```
-bash-4.1$ ./hello
Hello World, I am thread = 0
Number of threads = 12
Hello World, I am thread = 8
Hello World, I am thread = 3
Hello World, I am thread = 4
Hello World, I am thread = 6
Hello World, I am thread = 2
Hello World, I am thread = 9
Hello World, I am thread = 7
Hello World, I am thread = 1
Hello World, I am thread = 5
Hello World, I am thread = 10
Hello World, I am thread = 11
```

Executed the same program three times;
the order of outputs varies.

OpenMP Compiler Directives

Comprehensive information is available in OpenMP Specification

OpenMP Directive Format: C/C++

- ❑ **OpenMP directives**
  - start with #pragma omp
  - are case sensitive
  - apply to the succeeding statement (structured block)

```
#pragma omp directive-name [clause[ [,] clause]...] new-line
```

Parallel Construct

- **Creates a thread team and initiates parallel execution**
- `clause` **may be**
  - if(scalar-expression)
  - num_threads(integer-expression) // determines the number of threads
  - default(shared | none)
  - private(list)
  - firstprivate(list)
  - shared(list)
  - copyin(list)
  - reduction(redution-identifier :list)
  - proc_bind(master | close | spread)

```
#pragma omp parallel [clause[ [, ]clause] ...] new-line
                    structured-block
```

Worksharing Constructs

- ❑ **Share the work of executing a region**
  - each thread of the team executes a part
  - a worksharing construct is used in the context of `parallel` construct
- ❑ **Worksharing constructs**
  - loop
  - sections
  - single
- ❑ **Implicit barrier at the end of the worksharing construct**
  - `nowait` clause indicates that the implicit barrier may be omitted

Worksharing Constructs: Loop

- **Iterations of the loop are executed in parallel**
- *`clause`* **may be**
  - private(list)
  - firstprivate(list)
  - lastprivate(list)
  - reduction(reduction-identifier: list)
  - schedule(kind[, chunk_size]) // default chunk_size is 1
  - collapse(n)
  - ordered
  - nowait

```
#pragma omp for [clause[[,] clause] ... ] new-line
for-loops
```

Worksharing Constructs: Loop Schedule

- ❑ **schedule(static,** ***chunk_size*****)**
  - divides iterations into chunks of size chunk_size
  - assigns chunks to the threads in the team
  - round-robin fashion
- ❑ **schedule(dynamic,** ***chunk_size*****)**
  - schedules iterations based on thread requests
  - after the execution of a chunk the thread request the next chunk
- ❑ **schedule(guided,** ***chunk_size*****)**
  - similar to dynamic; but, the chunk size decreases during scheduling
- ❑ **schedule(auto)**
  - delegates scheduling decision to compiler or/and run-time system
- ❑ **schedule(runtime)**
  - defers the scheduling decision to run-time

Worksharing Constructs: Sections

- ❑ **A collection of structured blocks is executed in parallel**
- ❑ *clause* **may be**
  - private(list), firstprivate(list), lastprivate(list),
  - reduction(reduction-identifier:list),
  - nowait

```
#pragma omp sections [clause[[,] clause] ...] new-line
{
        [#pragma omp section new-line]
        structured-block
        [#pragma omp section new-line
        structured-block]
        ...
}
```

Worksharing Constructs: Single

- **Indicates that the structured block is executed by only one thread**
- *`clause`* **may be**
  - private(list)
  - firstprivate(list)
  - copyprivate(list)
  - nowait

```
#pragma omp single [clause[[,] clause] ...] new-line
structured-block
```

SIMD Construct

- **Specifies that the loop may be transformed into a SIMD loop**
  - execute loop iterations using SIMD instructions
  - SIMD instruction is a machine instruction that may operate on many data elements, SIMD stands for Single Instruction Multiple Data
  - *vector operation*: processes multiple pairs of operands in one step
- *clause* **may be**
  - safelen(length),
  - linear(list[:linear-step]), aligned(list[:alignment]),
  - private(list), lastprivate(list),
  - reduction(reduction-identifier:list), collapse(n)

```
#pragma omp simd [clause[[,] clause] ...] new-line
for-loops
```

Declare SIMD Construct

- ❑ **A function processes multiple arguments using SIMD instructions**

```
#pragma omp declare simd [clause[[,] clause] ...] new-line
[...]
function definition or declaration
```

```
void findmin (int *a, int *b, int *c) {
        #pragma omp simd
        for (i=0; i<N; i++)
        c[i] = min(a[i], b[i]);
}

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

Example credit: IBM

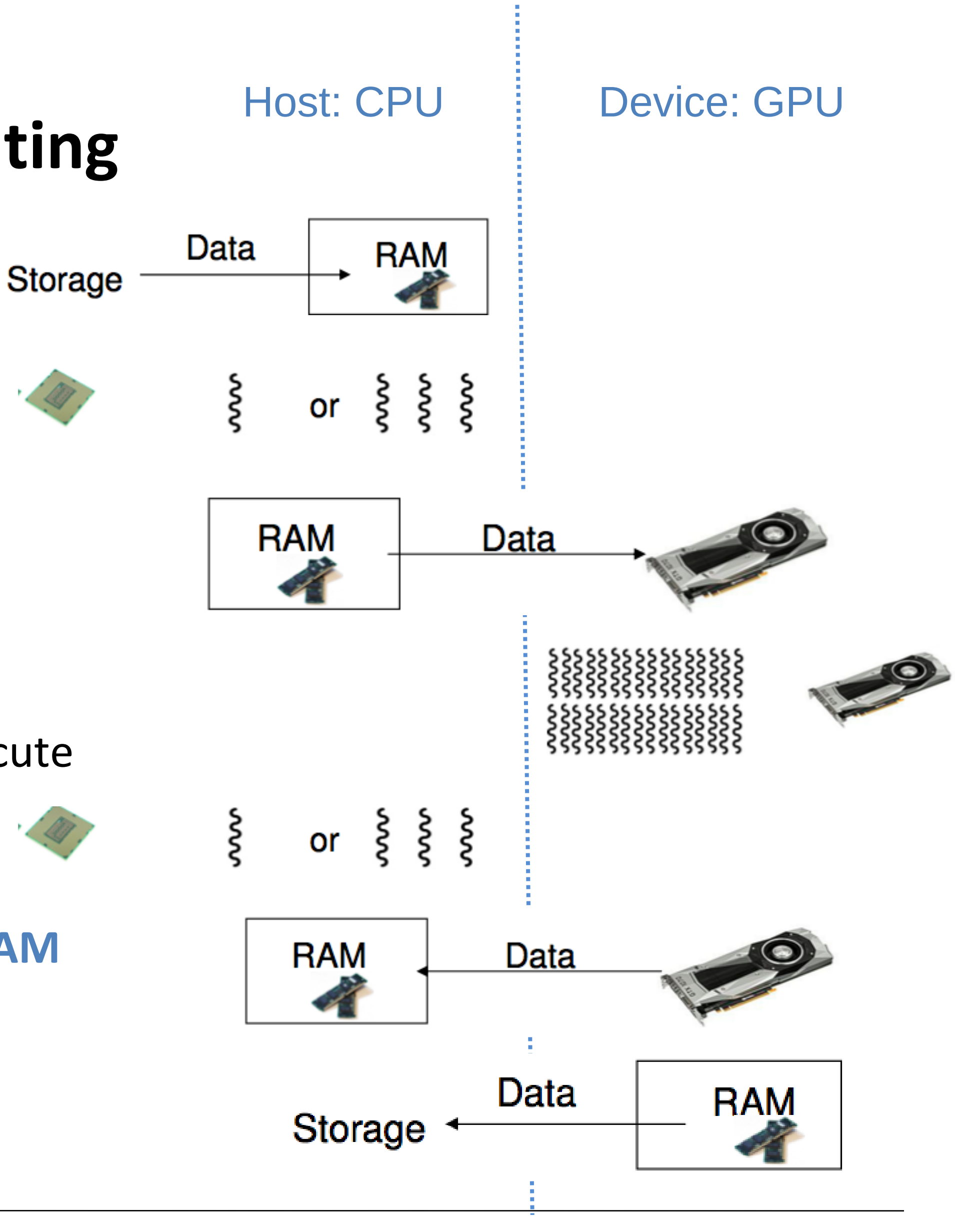
Loop SIMD Construct

- **Indicates a loop that may be executed using SIMD instructions**
  - loop iterations are executed in parallel also by team threads
  - *first the loop iterations are distributed across the team of threads, and thereafter chunks of iterations are transformed into SIMD loop*
- `clause` **may be**
  - any **`for`** or **`simd`** clause

```
#pragma omp for simd [clause[[,] clause] ...] new-line
for-loops
```

Heterogeneous Computing

- □ **Load data to the RAM**
- □ **Pre-process on CPU**
- □ **Load data to the GPU memory**
- □ **Process data with GPU**
  - CPU can either wait or execute some job
- □ **Copy data back from GPU to RAM**
- □ **Store result**



Device Constructs: target data

- **Creates the device data environment**
  - maps variables between *host* and *target device*
  - examples of *target devices*: **GPU**, Intel Xeon Phi,..
  - *host*: device that initiates OpenMP execution
  - typically the *host* is **CPU**
- *clause* **may be**
  - device (integer-expression)
  - map ([map-type :] list) // relates variables on host and target device
  - if (scalar-expression)

```
#pragma omp target data [clause[[,] clause],...] new-line
structured-block
```

Device Constructs: target

- ❑ **Creates the device data environment and initiates the execution on device**

```
#pragma omp target [clause[[,] clause],...] new-line
structured-block
```

```
#pragma omp target data
        map(to:a,b,c)
        map(from:e,f) {
        #pragma omp target {
                e = f(a,b,c);
        }
        #pragma omp target {
                f = g(a,b,c);
        }
}
```

Example credit: Intel

Device Constructs: target update

- **Ensures consistency between data on host and target device**

- *`clause`* **may be**
  - to(*list*)
  - from(*list*)
  - device(integer-expression)
  - if(scalar-expression)

```
#pragma omp target update [clause[[,] clause],...] new-line
```

Combined Constructs: Parallel Loop

- **Combined constructs are shortcuts for specifying constructs**
    - for instance: **parallel for**

- *`clause`* **may be**
    - any **`parallel`** or **`for`** clause

```
#pragma omp parallel for [clause[[,] clause] ...] new-line
for-loop
```

Tasking Constructs: task

- **Defines a task**
  - generates a task from *structured-block*
- *clause* **may be**
  - if(scalar-expression)
  - final(scalar-expression)
  - untied
  - default(shared | none)
  - mergeable
  - private(list), firstprivate(list)
  - shared(list), depend(dependence-type : list)

```
#pragma omp task [clause[[,] clause] ...] new-line
structured-block
```

Runtime Library Routines

Comprehensive information is available in OpenMP Specification

Examples of Library Routines

- **omp_set_num_threads(int num_threads)**
  - determines the number of threads
- **omp_get_num_threads(void)**
  - get the number of threads in the enclosing parallel region
- **omp_get_max_threads(void)**
  - indicates how many threads could be used
- **omp_get_thread_num(void)**
  - get the number of the thread (its ID) within the team
- **omp_get_num_procs(void)**
  - get the number of available processing elements
- **omp_set_nested(int nested)**
  - enables the nested parallelism

Examples of Lock Routines

- ❑ **OpenMP lock routines may be used for synchronization**
  - affect all tasks that call the routine
  - there are simple and nestable (may be set multiple times) lock routines
- ❑ **omp_init_lock(omp_lock_t *lock)**
  - initializes a simple OpenMP lock in the unlocked state
- ❑ **omp_set_lock(omp_lock_t *lock)**
  - suspends the task execution until the lock is available and it is set
- ❑ **omp_unset_lock(omp_lock_t *lock)**
  - unlocks the lock
- ❑ **omp_test_lock(omp_lock_t *lock)**
  - attempts to set an OpenMP lock without suspending the task execution
- ❑ **omp_destroy_lock(omp_lock_t *lock)**
  - deinitializes the lock

Examples of Timing Routines

- **omp_get_wtime(void)**
  - get the wall clock time in seconds; it is not consistent across all threads
- **omp_get_wtick(void)**
  - indicates the precision of the timer; number of seconds between clock ticks

```
double begin, end;

begin = omp_get_wtime();
       ...
end = omp_get_wtime();

printf(“Execution time: %f [s]\n", end - begin);
```

Environment Variables

Comprehensive information is available in OpenMP Specification

Examples of Environment Variables

- **Hello World example**
  - export OMP_NUM_THREADS=12
  - determines the number of threads used in parallel region
- **OMP_SCHEDULE**
  - affects the schedule type and chunk size of loop directives with schedule type runtime
  - for instance, export OMP_SCHEDULE "guided,12“
- **Thread affinity**
  - may improve the locality and memory access
  - OMP_PLACES: may be threads (hardware threads), cores, sockets (comprises one or more cores)
  - OMP_PROC_BIND: sets the thread affinity

Example: Matrix Multiplication

Matrix Multiplication

```
// initialization
…
begin = omp_get_wtime();
#pragma omp parallel for shared(a,b,c)
    for (int i = 0; i < dim; ++i) {
        for (int j = 0; j < dim; ++j) {
            for (int k = 0; k < dim; ++k) {
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
end = omp_get_wtime();
printf(“Execution time: %f [s]\n", end - begin);
…
```

Summary

- ❑ **We have highlighted some features of OpenMP 4.0**
  - ▪ more details are provided in the corresponding specification
- ❑ **Various compilers support various versions of OpenMP**
  - ▪ OpenMP 4.0 is supported by GCC 4.9 or later version
  - ▪ OpenMP 3.0 is supported by GCC 4.4 or later version
  - ▪ not everything that is described in specification is supported by a certain compiler
- ❑ **Currently in our lab is installed**
  - ▪ gcc version 5.4.0