An Introduction to OpenMP

Helen He
Lawrence Berkeley National Lab

Oscar Hernandez
Oak Ridge National Lab

***Acknowledgements**: Thanks to Tim Mattson, Barbara Chapman, Michael Klemm, and many others for the materials used in this presentation*

Agenda

- Part 1 (ca. 65 min):
  - **Background**
  - **Parallel Regions**
  - **Working with Loops**
  - **Data Environment**
  - **Tasks**
  - **Thread Affinity**

Presenter: **Helen He**

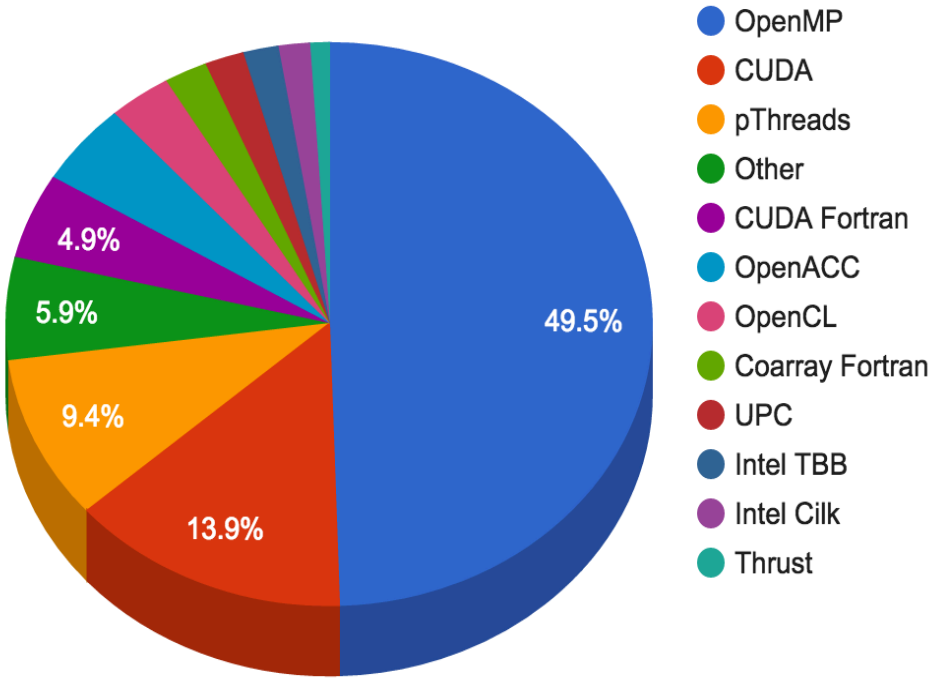
- Part 2 (ca. 45 min)
  - **Using GPUs**
  - **SIMD Features**
  - **OpenMP 5.0 Topics**
  - **Wrap Up / More Info**

Presenter: **Oscar Hernandez**

Choice of programming models for modern HPC systems

- MPI was developed primarily for inter-address space (inter means between or among)
- OpenMP was developed for **shared memory** or intra-node (intra means within)
  - Trend: multi-socket nodes with rapidly increasing core counts
  - Trend: accelerators
- Hybrid Programming (**MPI+X**) is when we use a solution with different programming models for inter *vs.* intra-node parallelism

What is X in "MPI+X"? OpenMP is about 50% in 2015 at NERSC

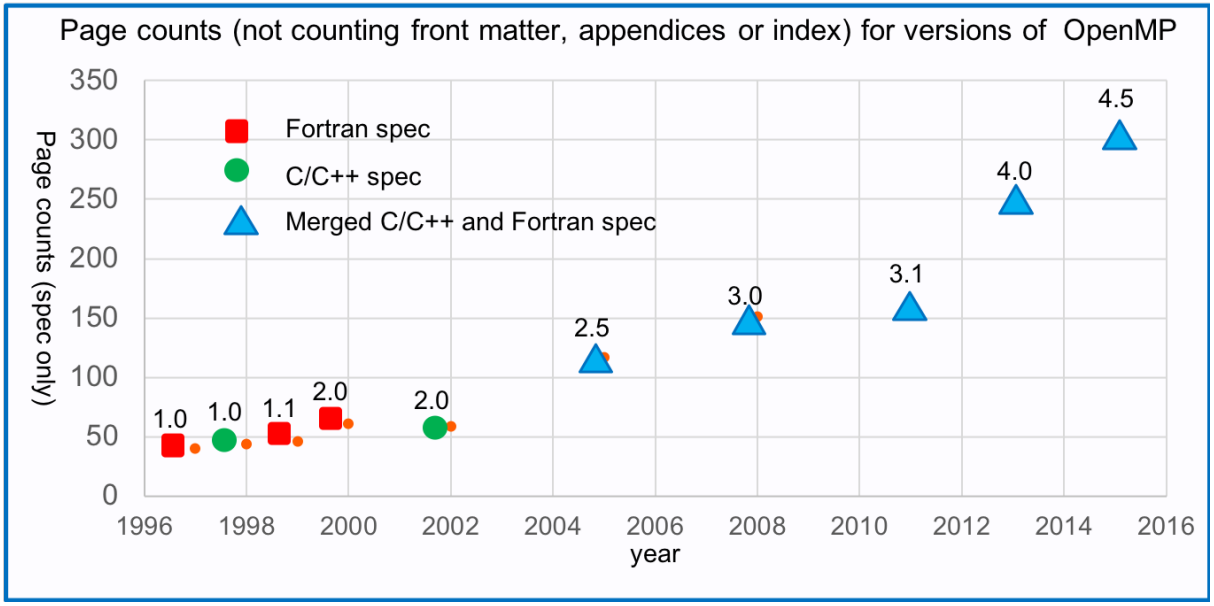


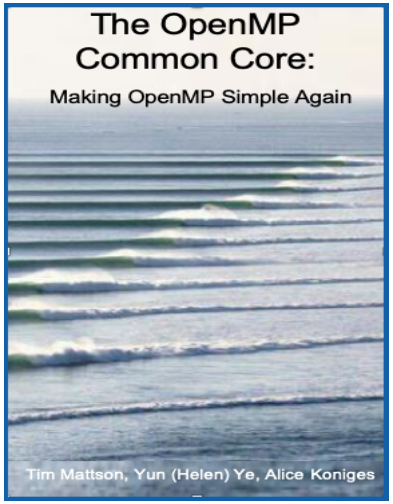
What is OpenMP

- OpenMP is an industry standard API of C, C++, and Fortran for shared memory, vectorization, and heterogeneous device programming
  - Specifications by OpenMP Architecture Review Board (ARB)
    - 30+ members including industry, government labs, and academia
    - Available in major commercial and open source compilers
  - The ARB mission is **to standardize directive-based multi-language high-level parallelism that is performant, productive, and portable**
    - Multi-level parallelism supported (coprocessors, threads, SIMD)
    - Task-based programming model is a modern approach to parallelism

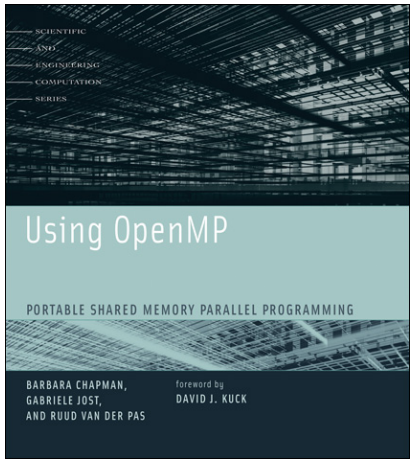
- OpenMP Resources: https://www.openmp.org
  - Specifications, Reference Guides, Examples
  - Tutorials, blogs, compilers and tools

Growth of OpenMP

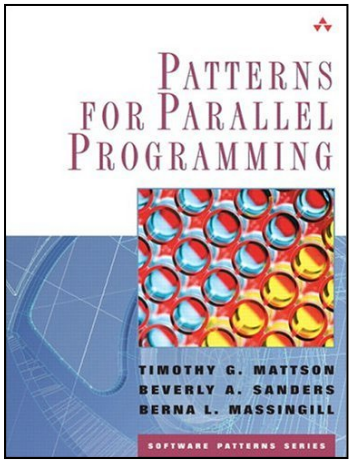




An upcoming book about OpenMP Common Core, 2019 (cover image TBD)



A book about OpenMP by a team of authors at the forefront of OpenMP's evolution, 2010



A book about how to "think parallel" with examples in OpenMP, MPI and java, 201x

A new book that covers OpenMP features beyond OpenMP 2.5, 2017

OpenMP On Your Laptop (1)

- Mac OS X/Sierra
  - Compiler installed in native XCode doesn't "do" OpenMP
  - Install "homebrew"
    - https://brew.sh/
  - Install GNU Compiler Collection
    - $ brew update
    - $ brew install gcc **(this will take a long time)**
  - Use the compilers! **(note the command names)**
    - $ gcc-9 -fopenmp program.c
    - $ g++-9 -fopenmp program.cpp
    - $ gfortran-9 -fopenmp program.f90
- Windows
  - Install free version of Visual Studio
    - https://www.visualstudio.com/downloads/

OpenMP On Your Laptop (2)

- Linux
  - Install GNU Compiler Collection (C, C++, Fortran)
    - Ubuntu-like: C, C++ already there by default
      - # apt install gfortran
    - CentOS-like
      - # yum install gcc-c++ gcc-gfortran
    - Fedora-like
      - # dnf install gcc-c++ gcc-gfortran
  - Use compilers!
    - $ gcc -fopenmp program.c
    - $ g++ -fopenmp program.cpp
    - $ gfortran -fopenmp program.f90

Hands-on Exercises

- Slides and exercises will be posted at:
  - https://bluewaters.ncsa.illinois.edu/bw-petascale-computing-2019/resources
- Exercises with detailed README are also available on:
  - NERSC Cori: /project/projectdirs/training/OpenMP_PCI2019
  - NCSA Blue Waters: /u/training/instr039/OpenMP_PCI2019
  - Compilers: Intel, Cray, GCC, PGI
- Please follow the instructions to do these 7 exercises after the tutorial.
- Always try to work on writing your own programs before looking at the answers.
- You can also work on other HPC systems, such as TACC Stampede2, or run on your own laptop.

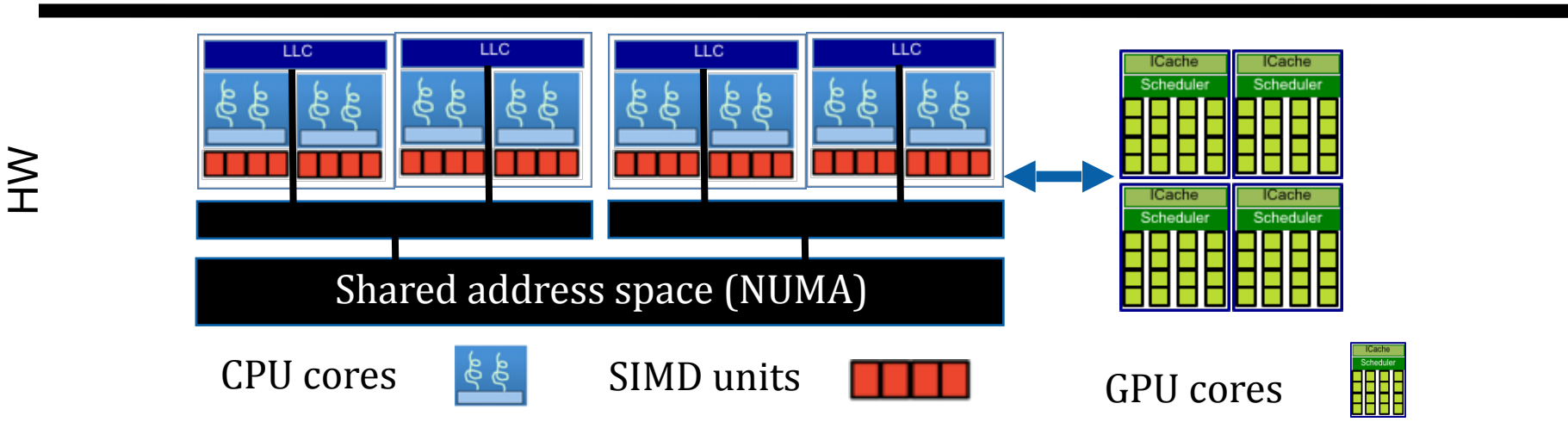
How Does OpenMP Work?

- Teams of OpenMP threads are created to perform the computation in a code
  - **Work is divided** among the threads, which run on the different cores
  - The threads collaborate **by sharing variables**
  - Threads **synchronize** to order accesses and prevent data corruption
  - **Structured programming** is encouraged to reduce likelihood of bugs

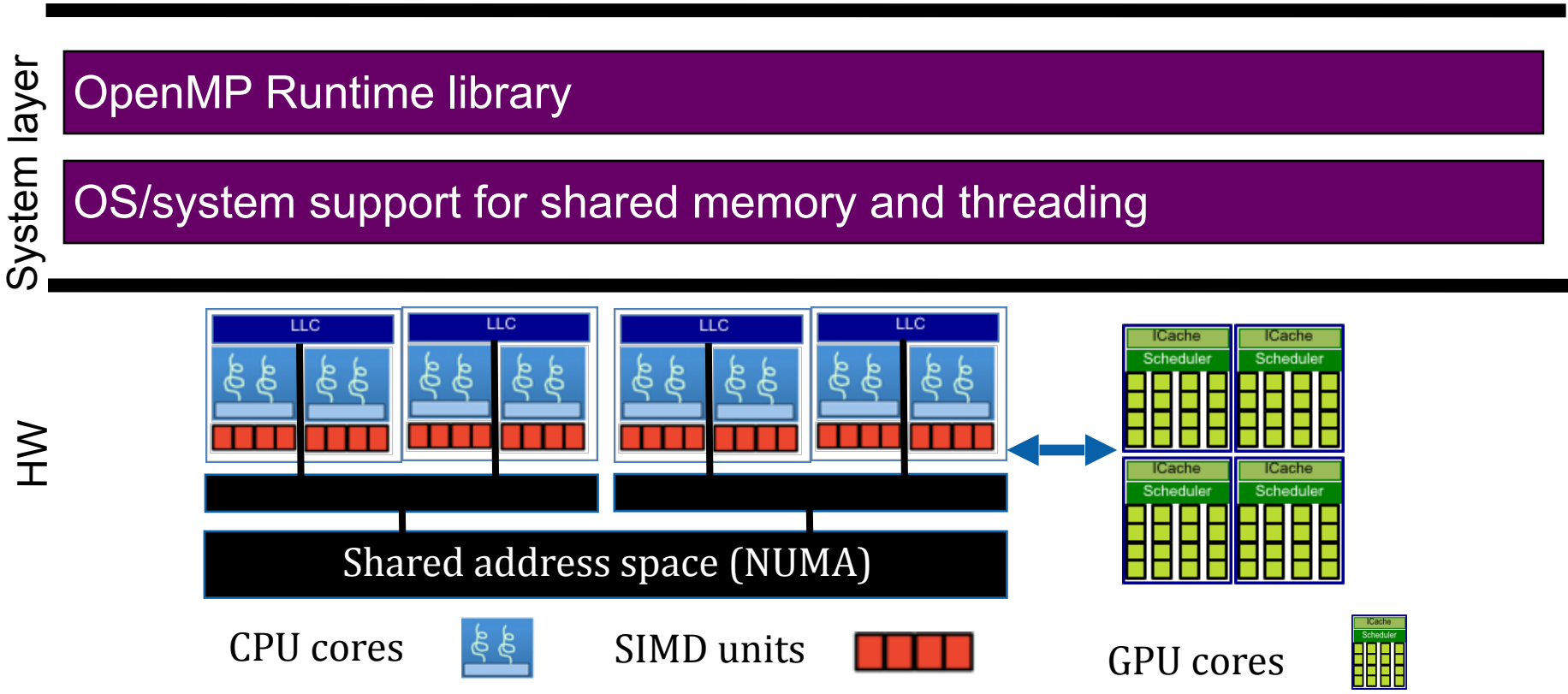
- OpenMP components:
  - Compiler Directives and Clauses
  - Runtime Libraries
  - Environment Variables

OpenMP Basic Definitions: Basic Solution Stack

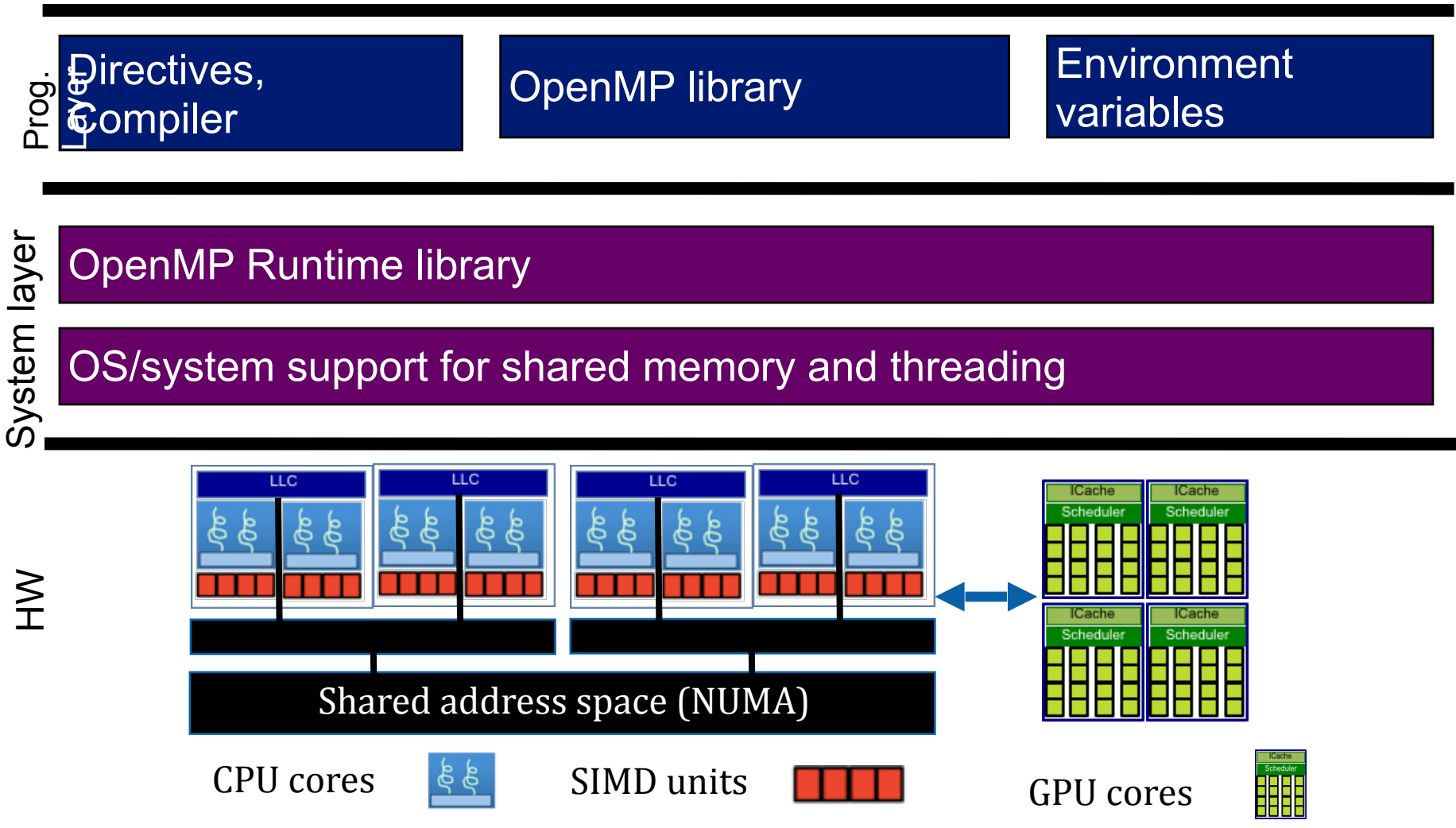
OpenMP Basic Definitions: Basic Solution Stack



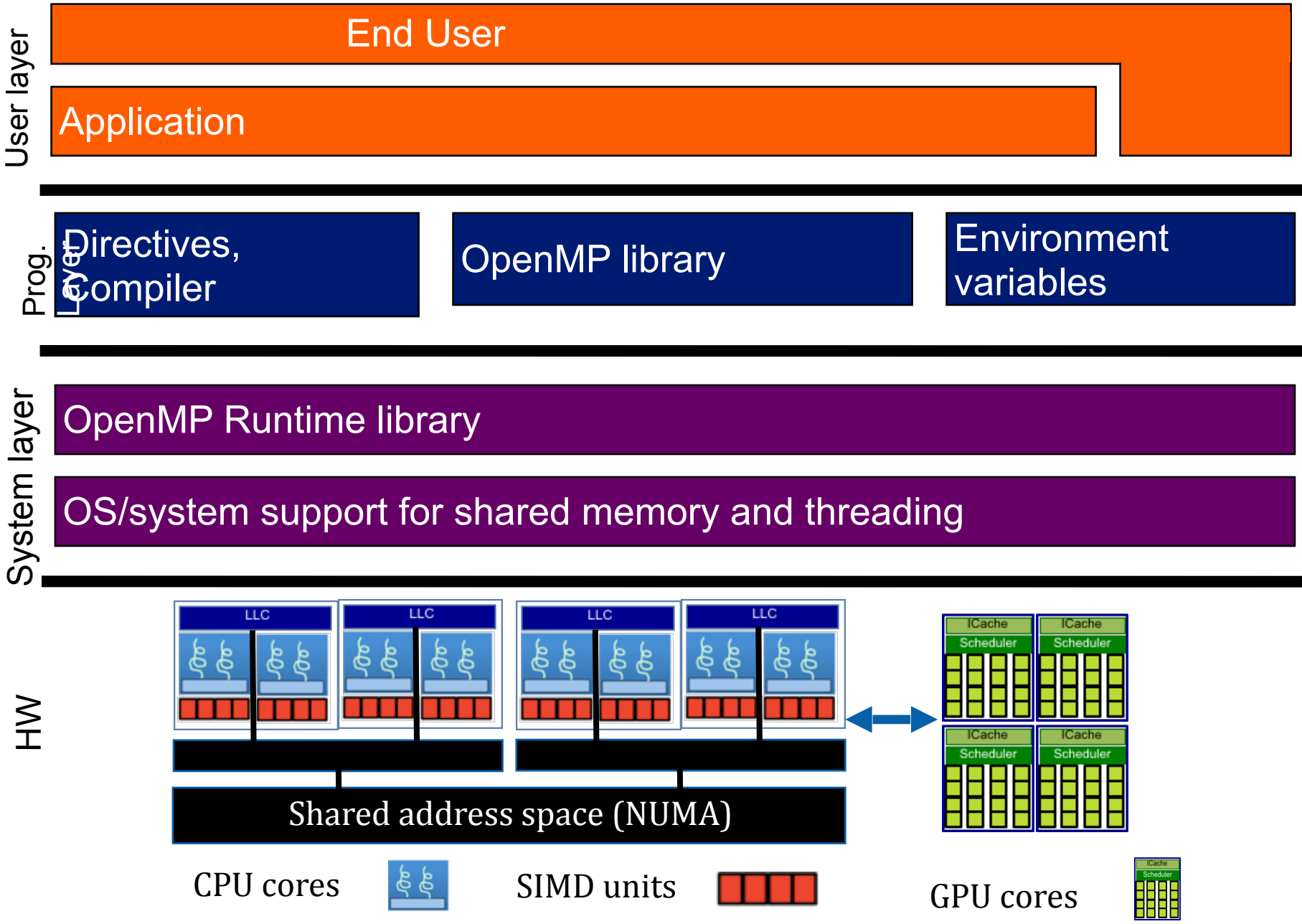
OpenMP Basic Definitions: Basic Solution Stack



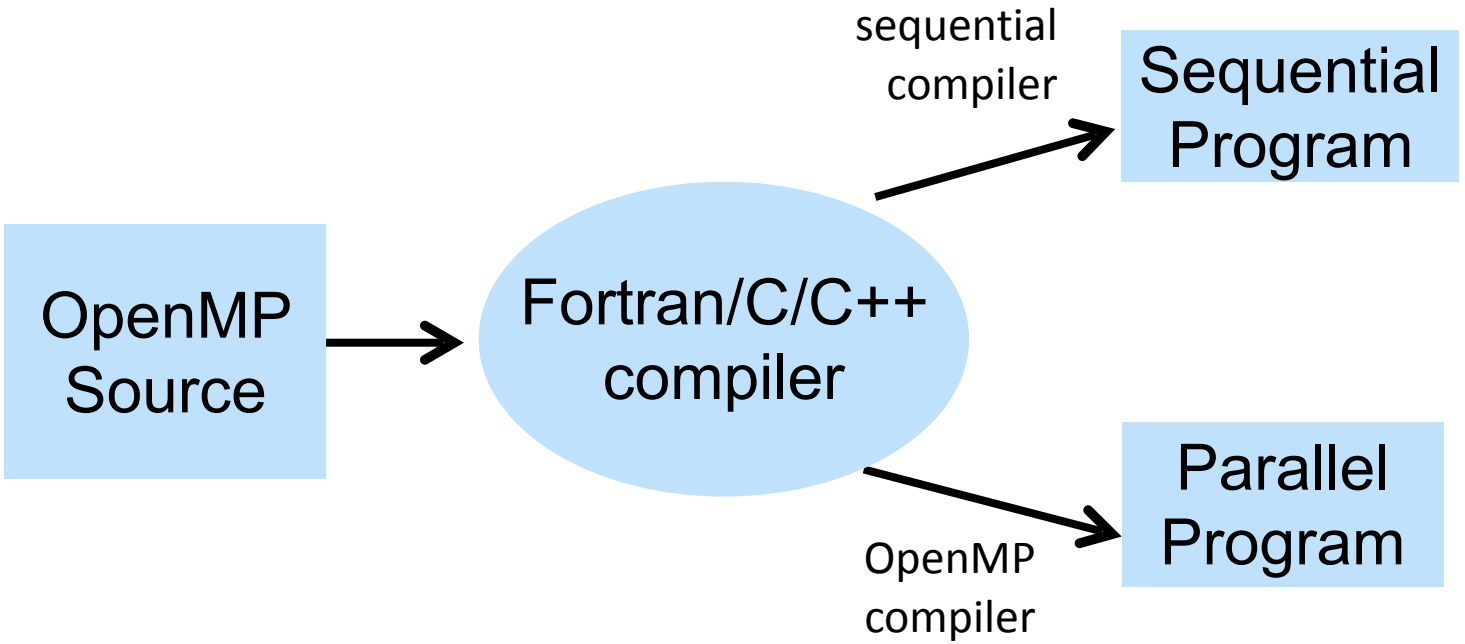
OpenMP Basic Definitions: Basic Solution Stack



OpenMP Basic Definitions: Basic Solution Stack



OpenMP Usage

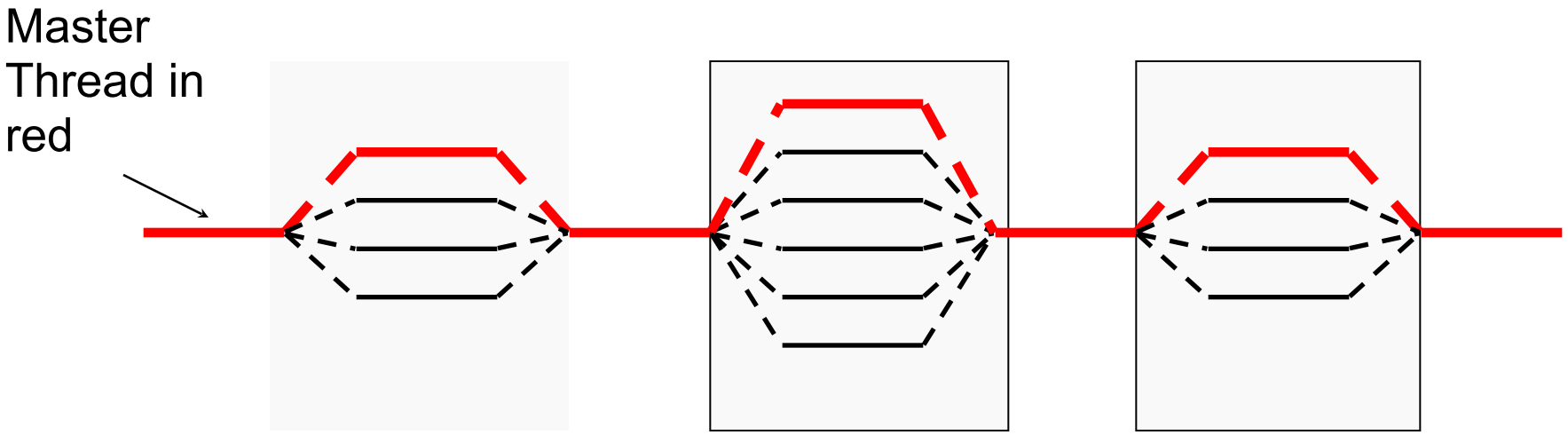


- Maintains single source code for serial and parallel programs.
- Needs special flag to enable OpenMP, such as:
  - Intel compiler: -qopenmp
  - GNU compiler: -fopenmp
  - PGI compiler: -mp
  - Cray compiler: none

OpenMP Programming Model

Fork-Join Parallelism:

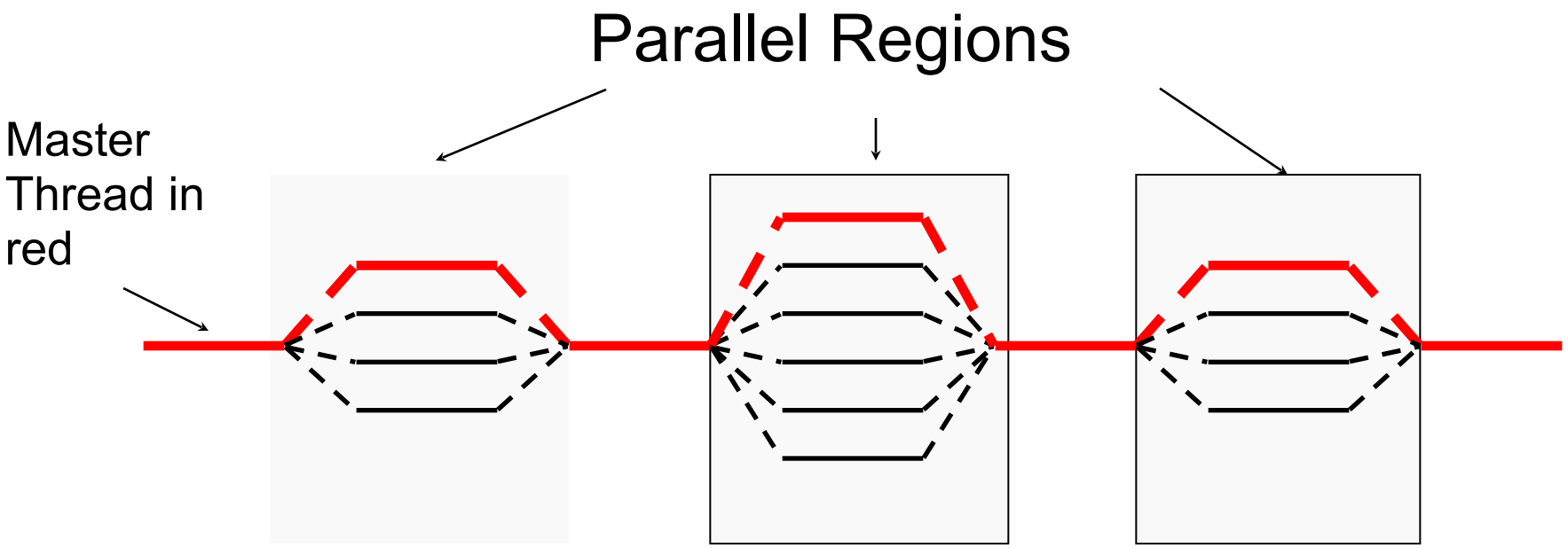
- Master thread spawns a team of threads as needed.
- Parallelism added incrementally until performance goals are met, i.e., the sequential program evolves into a parallel program.



OpenMP Programming Model

Fork-Join Parallelism:

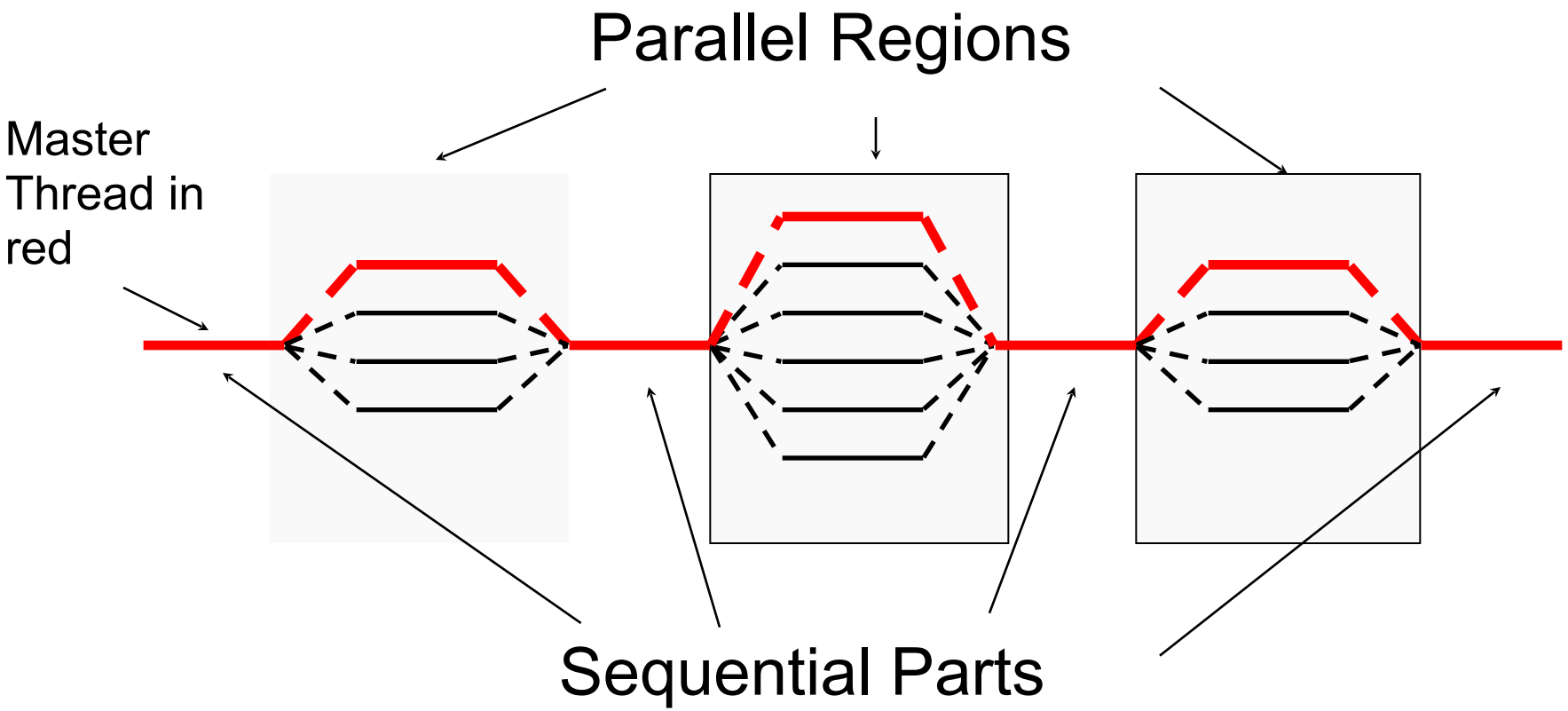
- Master thread spawns a team of threads as needed.
- Parallelism added incrementally until performance goals are met, i.e., the sequential program evolves into a parallel program.



OpenMP Programming Model

Fork-Join Parallelism:

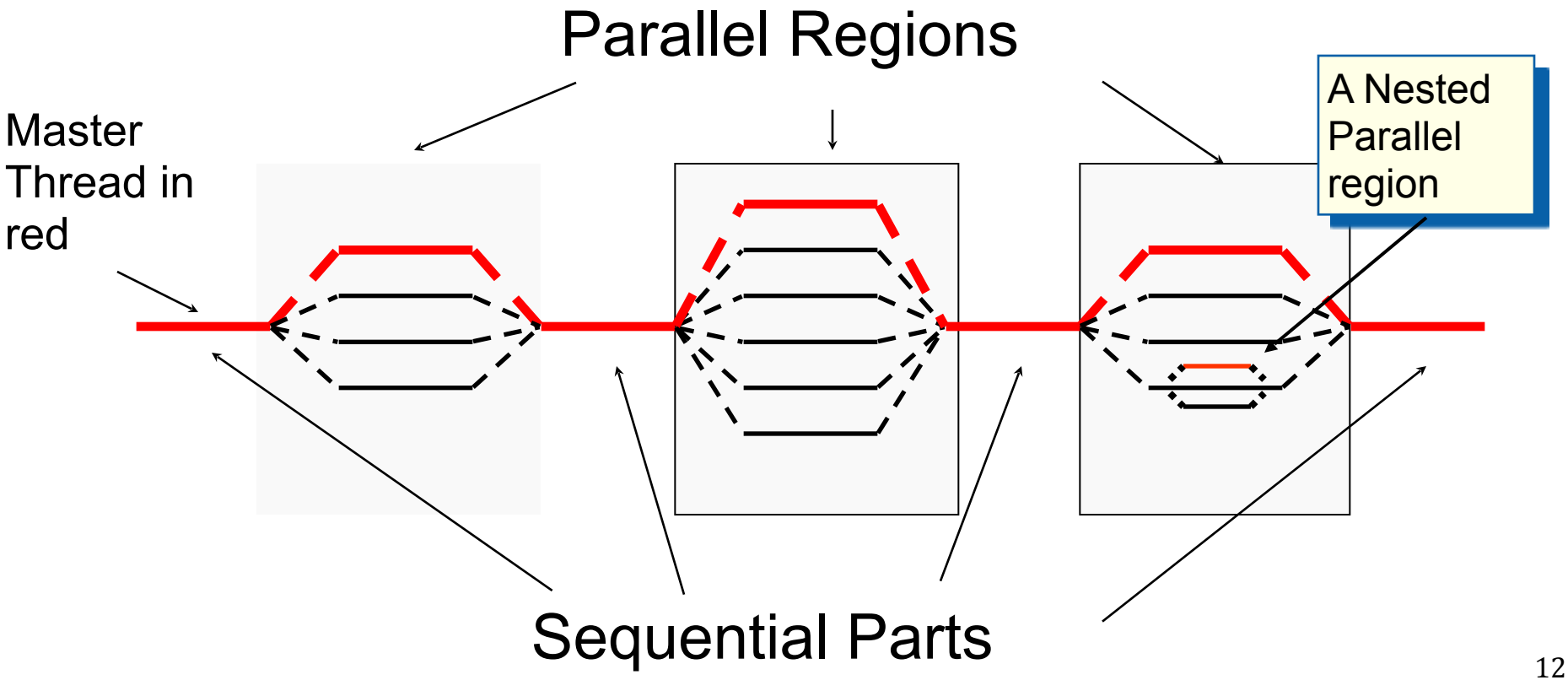
- Master thread spawns a team of threads as needed.
- Parallelism added incrementally until performance goals are met, i.e., the sequential program evolves into a parallel program.



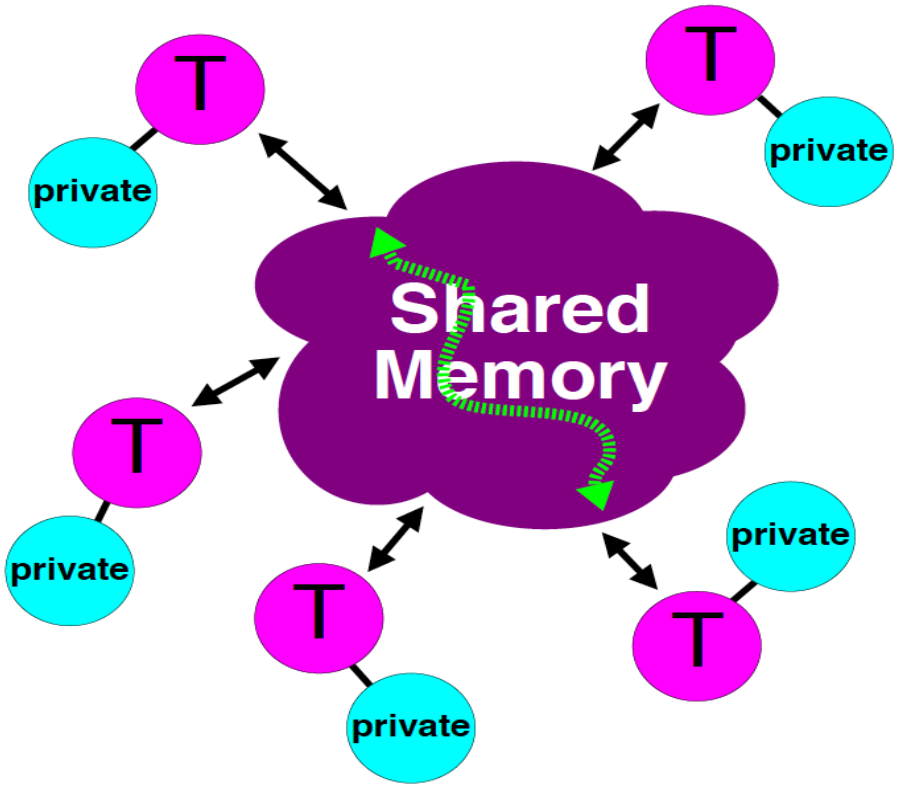
OpenMP Programming Model

Fork-Join Parallelism:

- Master thread spawns a team of threads as needed.
- Parallelism added incrementally until performance goals are met, i.e., the sequential program evolves into a parallel program.



OpenMP Memory Model



- ✔ *All threads have access to the same, globally shared, memory*
- ✔ *Data can be shared or private*
- ✔ *Shared data is accessible by all threads*
- ✔ *Private data can only be accessed by the thread that owns it*
- ✔ *Data transfer is transparent to the programmer*
- ✔ *Synchronization takes place, but it is mostly implicit*

Serial *vs.* OpenMP

Serial

```
void main ()
{
   double x(256);
   for (int i=0; i<256; i++)
       {
       some_work(x[i]);
       }
}
```

OpenMP

```
#include "omp.h"
Void main ()
{
   double x(256);
#pragma omp parallel for
   for (int i=0; i<256; i++)
       {
       some_work(x[i]);
       }
}
```

OpenMP is not just parallelizing loops! It offers a lot more ….

Advantages of OpenMP

- Simple programming model
  - Data decomposition and communication handled by compiler directives
- Single source code for serial and parallel codes
- No major overwrite of the serial code
- Portable implementation
- Progressive parallelization
  - Start from most critical or time consuming part of the code

OpenMP Basic Syntax

- Fortran: case insensitive
  - Add: use omp_lib or include “omp_lib.h”
  - Fixed format
    - Sentinel directive *[clauses]*
    - Sentinel could be: !$OMP, *$OMP, c$OMP
  - Free format
    - !$OMP directive *[clauses]*
- C/C++: case sensitive
    - Add: #include “omp.h”
    - #pragma omp directive *[clauses] newline*

Compiler Directives

- Parallel construct
  - Fortran: PARALLEL …. END PARALLEL
  - C/C++: paralllel
- Worksharing constructs
  - Fortran: DO … END DO, WORKSHARE
  - C/C++: for
  - Both: sections
- Synchronization
  - Barrier, critical, atomic, master, single, ordered, flush, ..
- Tasking
  - task, taskwait, taskgroup, taskloop, …
- SIMD
  - simd, declare simd, …
- Device
  - Target data, target , teams distribute, …

Clauses

- private (list), shared (list)
- firstprivate (list), lastprivate (list)
- reduction (operator: list)
- schedule (method *[,chunk_size]*)
- nowait
- if (scalar_expression)
- num_threads (num)
- threadprivate (list), copyin (list)
- ordered
- collapse (n)
- tie, untie
- copyin
- map

- And more ….

Runtime Libraries

- Number of threads: omp_{set,get}_num_threads
- Thread ID: omp_get_thread_num
- Scheduling: omp_{set,get}_dynamic
- Nested parallelism: omp_in_parallel
- Locking: omp_{init,set,unset}_lock
- Active levels: omp_get_thread_limit
- Wallclock Timer: omp_get_wtime
  - Thread private
  - Call function twice, use difference between end time and start time
- And more …

Environment Variables

- OMP_NUM_THREADS
- OMP_SCHEDULE
- OMP_STACKSIZE
- OMP_DYNAMIC
- OMP_NESTED
- OMP_WAIT_POLICY
- OMP_ACTIVE_LEVELS
- OMP_THREAD_LIMIT
- OMP_DISPLAY_ENV
- OMP_DISPLAY_AFFINITY
- And more ….

A Simple OpenMP Program

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main () {
    int tid, nthreads;
#pragma omp parallel private(tid)
   {
    tid = omp_get_thread_num();
    printf("Hello World from thread %d\n", tid);
#pragma omp barrier
     if ( tid == 0 ) {
        nthreads = omp_get_num_threads();
        printf("Total threads= %d\n",nthreads);
     }
  }
}
```

Sample Compile and Run:

```
% ifort -qopenmp test.f90
% export OMP_NUM_THREADS=4
% ./a.out
```

```
Program main
use omp_lib       (or: include "omp_lib.h")
integer :: id, nthreads
!$OMP PARALLEL PRIVATE(id)
    id = omp_get_thread_num()
    write (*,*) "Hello World from thread", id
!$OMP BARRIER
    if ( id == 0 ) then
        nthreads = omp_get_num_threads()
        write (*,*) "Total threads=",nthreads
     end if
!$OMP END PARALLEL
End program
```

Sample Output: (no specific order)

```
Hello World from thread          0
Hello World from thread          2
Hello World from thread          3
Hello World from thread          1
Total threads=       4
```

Sample Nested OpenMP Program

```
#include <omp.h>
#include <stdio.h>
void report_num_threads(int level)
{
    #pragma omp single {
        printf("Level %d: number of threads in the
team: %d\n", level, omp_get_num_threads());
        }
}
int main()
{
    omp_set_dynamic(0);
    #pragma omp parallel num_threads(2) {
        report_num_threads(1);
        #pragma omp parallel num_threads(2) {
            report_num_threads(2);
            #pragma omp parallel num_threads(2) {
                report_num_threads(3);
            }
        }
    }
    return(0);
}
```

```
% a.out
Level 1: number of threads in the team: 2
Level 2: number of threads in the team: 1
Level 3: number of threads in the team: 1
Level 2: number of threads in the team: 1
Level 3: number of threads in the team: 1
```

```
% export OMP_NESTED=true
% export OMP_MAX_ACTIVE_LEVELS=3
% a.out
Level 1: number of threads in the team: 2
Level 2: number of threads in the team: 2
Level 2: number of threads in the team: 2
Level 3: number of threads in the team: 2
Level 3: number of threads in the team: 2
Level 3: number of threads in the team: 2
Level 3: number of threads in the team: 2
```

```
Level 0: P0
Level 1: P0 P1
Level 2: P0 P2; P1 P3
Level 3: P0 P4; P2 P5; P1 P6; P3 P7
```

Thread Creation: the parallel Construct

FORTRAN:

```
!$OMP PARALLEL PRIVATE(id)
   id = omp_get_thread_num()
   write (*,*) "I am thread", id
!$OMP END PARALLEL
```

C/C++:

```
#pragma omp parallel private(thid)
{
    thid = omp_get_thread_num();
    printf("I am thread %d\n",
    thid);
}
```

- **A team of threads for parallel execution can only be created with the parallel construct.**
- **Each thread executes codes within the OpenMP parallel region.**
- **Threads wait at the end of parallel construct until all threads are finished with the parallel region before any proceed past the end of the parallel region.**

if and num_threads Clauses

C/C++ example:

```
    int n=some_func();

#pragma omp parallel if(n>10)
    {
      … do_stuff;
     }
```

C/C++ example:

```
    int n=some_func();
#pragma omp parallel num_threads(n)
    {
      … do_stuff;
     }
```

- **The if clause contains a conditional expression. Fork only occurs if it is TRUE.**
- **The num_threads defines the number of threads active in a parallel construct**

Various Methods to Set #threads

1) Use num_threads clause

```
#pragma omp parallel num_threads (4)
{
    int ID = omp_get_thread_num();
    pooh(ID,A);
}
```

2) Call omp_set_num_threads API

```
omp_set_num_threads(4);
#pragma omp parallel
{
   int ID = omp_get_thread_num();
   pooh(ID,A);
}
```

3) Set runtime environment

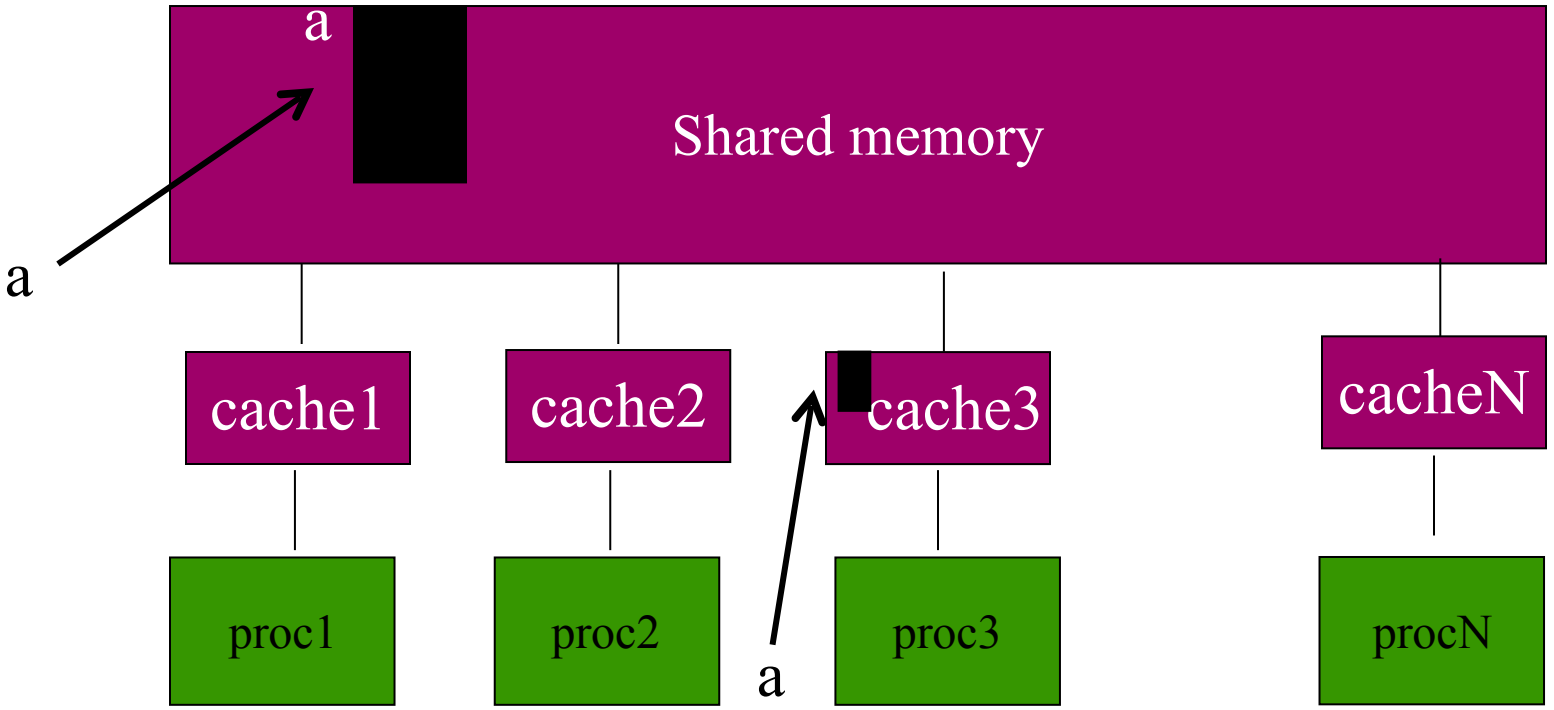
```
export OMP_NUM_THREDS=4
#pragma omp parallel
{
   int ID = omp_get_thread_num();
   pooh(ID,A);
}
```

4) Do none of the three above.
Code will use an implementation dependent default number of threads defined by the compiler.

- **Precedence: 1) > 2) > 3) > 4)**
- **You may get fewer threads than you requested, check with omp_get_num_threads()**

OpenMP and Relaxed Consistency

- OpenMP supports a **relaxed-consistency** shared memory model
  - Threads can maintain a **temporary view** of shared memory that is not consistent with that of other threads
  - These temporary views are made consistent only at certain points in the program
  - The operation that enforces consistency is called the **flush operation**



Synchronization

- Synchronization is used to impose order constraints and to protect access to shared data

- High level synchronization:
  - **critical**
  - **barriel**
  - atomic
  - Ordered

- Low level synchronization
  - flush
  - locks (both simple and nested)

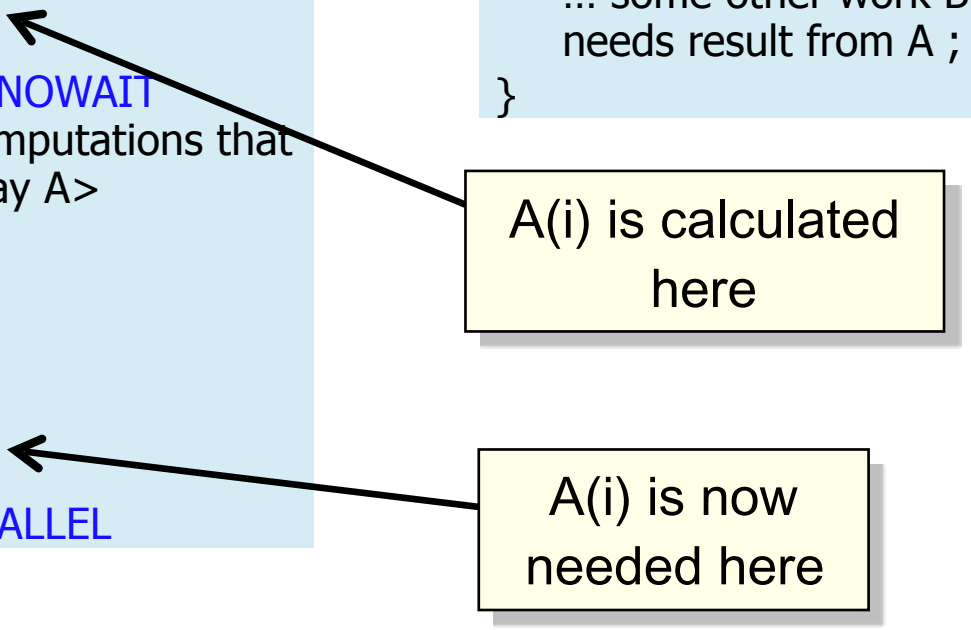
Synchronization: barrier Directive

FORTRAN:

```
!$OMP PARALLEL
!$OMP DO
do i = 1, n
   A(i)=B(i)+C(i)
enddo
!$OMP END DO NOWAIT
<some other computations that
do not need Array A>
!$OMP BARRIER
!$OMP DO
do i = 1, n
   E(i)=A(i)*D(i)
enddo
!$OMP END DO
!$OMP END PARALLEL
```

C/C++:

```
#pragma omp parallel
{   … some work A;
    #pragma omp barrier
    … some other work B that
    needs result from A ;
}
```



- **Every thread will wait until all threads at the barrier.**
- **Barrier makes sure all the shared variables are (explicitly) synchronized.**

Synchronization: critical Directive

FORTRAN:

```
!$OMP PARALLEL SHARED (x)
     … some work …
!$OMP CRITICAL [name]
    x =x + 1.0
!$OMP END CRITICAL
   … some other work …
!$OMP END PARALLEL
```

C/C++:

```
#pragma omp parallel shared (x)
{
#pragma omp critical
  {
  x = x +1.0;
  }
}
```

- **Every thread executes the critical region one at a time.**
- **Multiple critical regions with no name are considered as one critical region: single thread execution at a time.**

Flush and Synchronization

- A flush operation is implied by OpenMP synchronizations, e.g.,
  - at entry/exit of parallel regions
  - at implicit and explicit barriers
  - at entry/exit of critical regions

  ….

  (but not at entry to worksharing regions)

Do not mix reads and writes of a variable across multiple threads, you cannot assume the reading threads see the results of the writes. It may cause data race.

Synchronization: master and single Directives

FORTRAN:
```
!$OMP MASTER
    ... some work ...
!$OMP END MASTER
```

C/C++:
```
#pragma omp master
{
   ... some work ...
}
```

FORTRAN:
```
!$OMP SINGLE
    ... some work ...
!$OMP END SINGLE
```

C/C++:
```
#pragma omp single
{
   ... some work ...
}
```

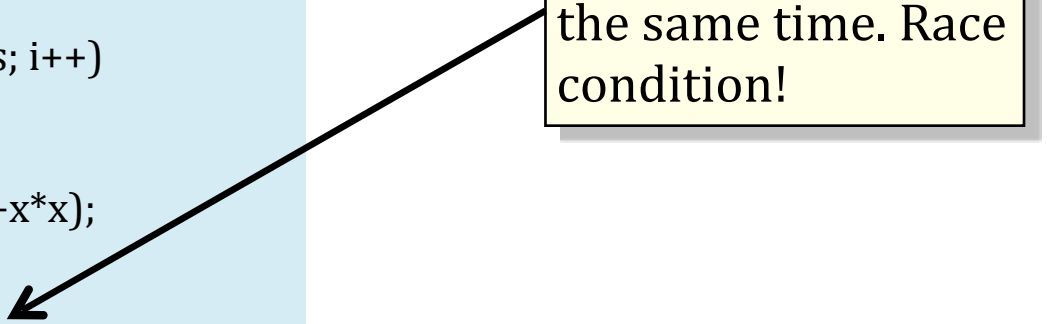
- **Master region:**
  - **Only the master threads executes the MASTER region.**
  - **No implicit barrier at the end of the MASTER region.**
- **Single region:**
  - **First thread arrives the SINGLE region executes this region.**
  - **All threads wait: implicit barrier at end of the SINGLE region.**

Thread Safety

- In general, IO operations, general OS functionality, common library functions may not be thread safe. They should be performed by one thread only or serialized.
- Avoid race condition in OpenMP program.
  - Race condition: Multiple threads are updating the same shared variable simultaneously.
  - Use “critical” directive
  - Use “atomic” directive
  - Use “reduction” directive

pi_omp_wrong.c

```
/* pi_omp_wrong.c */

...
int main()
{
   int i; double x, pi, sum = 0.0;
   step = 1.0/(double) num_steps;

#pragma omp parallel private(x, sum)
{
#pragma omp for
   for (i=0; i<=num_steps; i++)
   {
     x=(i+0.5)*step;
     sum=sum+ 4.0/(1.0+x*x);
   }
   pi = pi + step * sum;
}
   printf("pi=%f\n",pi);
   return 0;
}
```



Pi Program with False Sharing*

- Original Serial pi program with 100000000 steps ran in 1.83 seconds.

Example: A simple Parallel pi program

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{          int i, nthreads;  double pi, sum[NUM_THREADS];
           step = 1.0/(double) num_steps;
           omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel
   {
           int i, id,nthrds;
          double x;
          id = omp_get_thread_num();
          nthrds = omp_get_num_threads();
          if (id == 0)  nthreads = nthrds;
           for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                    x = (i+0.5)*step;
                    sum[id] += 4.0/(1.0+x*x);
           }
   }
           for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i] * step;
}
```

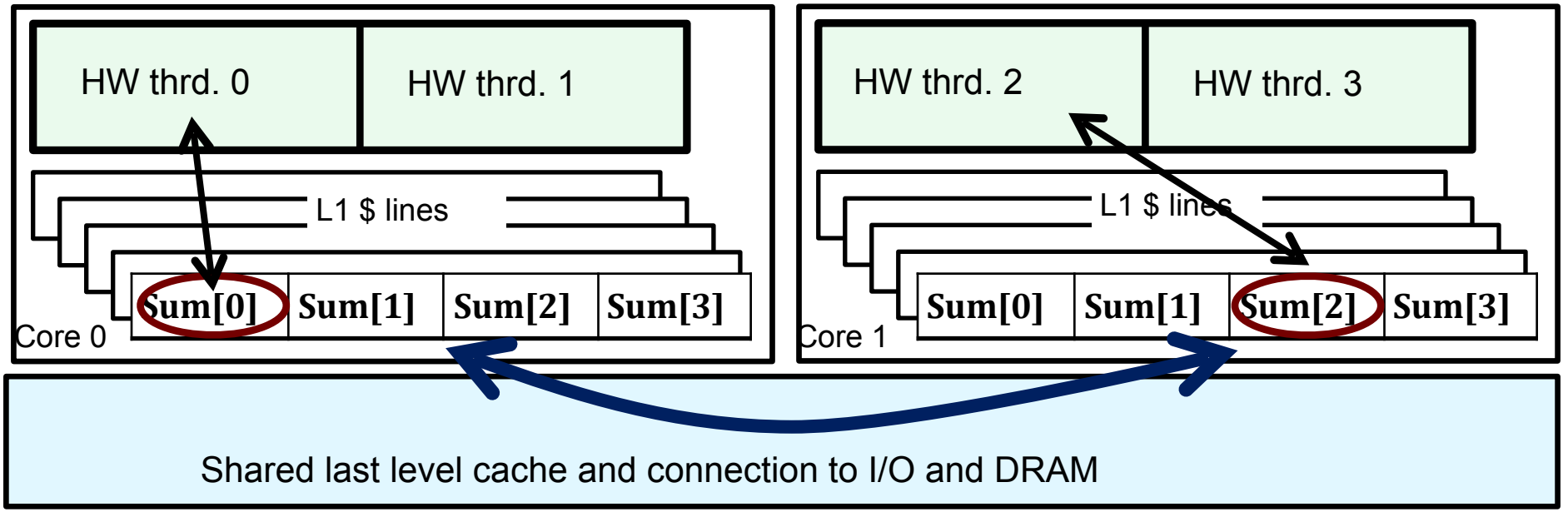
Promoting sum to an array made the coding easy, but led to false sharing and poor performance.

threads	1st SPMD
1	1.86
2	1.03
3	1.08
4	0.97

*Intel compiler (icpc) with default optimization level (O2) on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

Why Such Poor Scaling? False Sharing

- If independent data elements happen to sit on the same cache line, each update will cause the cache lines to "slosh back and forth" between threads … This is called **"false sharing"**.



- If you promote scalars to an array to support creation of an SPMD program, the array elements are contiguous in memory and hence share cache lines … Results in poor scalability.
- Solution: Pad arrays so elements you use are on distinct cache lines.

Eliminate False Sharing via Padding

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define    PAD      8          // assume 64 byte L1 cache line size
#define NUM_THREADS 2
void main ()
{               int i, nthreads;  double pi, sum[NUM_THREADS][PAD];
                step = 1.0/(double) num_steps;
                omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel
   {            int i, id,nthrds;
            double x;
            id = omp_get_thread_num();
            nthrds = omp_get_num_threads();
            if (id == 0)   nthreads = nthrds;
                for (i=id, sum[id][0]=0.0;i< num_steps; i=i+nthrds) {
                            x = (i+0.5)*step;
                            sum[id][0] += 4.0/(1.0+x*x);
                }
    }
                for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i][0] * step;
}
```

Pad the array so each sum value is in a different cache line

threads	1st SPMD	1st SPMD padded
1	1.86	1.86
2	1.03	1.01
3	1.08	0.69
4	0.97	0.53

Pi with critical

```
/* file name: pi_omp_critical.c */
...
int main()
{
   int i; double x, pi, sum = 0.0;
   step = 1.0/(double) num_steps;

#pragma omp parallel private(x, sum)
{
#pragma omp for
   for (i=0; i<=num_steps; i++)
   {
     x=(i+0.5)*step;
     sum=sum+ 4.0/(1.0+x*x);
   }
#pragma omp critical
   pi = pi + step * sum;
}
   printf("pi=%f\n",pi);
   return 0;
}
```

Create a scalar local to each thread to accumulate partial sums. No array, no false sharing.

"critical" ensures updates are performed one thread at a time

Loop-based vs. SPMD

Loop-based:

```
!$OMP PARALLEL DO PRIVATE(i)
!$OMP&            SHARED(a,b,n)
      do i=1, n
        a(i) = a(i) + b(i)
    enddo
!$OMP END PARALLEL DO
```

SPMD (Single Program Multiple Data):

```
!$OMP PARALLEL DO PRIVATE(start, end, i)
!$OMP&                  SHARED(a,b)
    num_thrds = omp_get_num_threads()
    thrd_id = omp_get_thread_num()
    start = n * thrd_id/num_thrds + 1
    end = n * (thrd_num+1)/num_thrds
    do i = start, end
       a(i) = a(i) + b(i)
    enddo
!$OMP END PARALLEL DO
```

SPMD code normally gives better performance than loop-based code, but is more difficult to implement:

- **Less thread synchronization.**
- **Less cache misses.**
- **More compiler optimizations.**

Worksharing-Loop Parallelism

FORTRAN:

```
!$OMP PARALLEL [Clauses]
  ...
!$OMP DO [Clauses]
      do i = 1, 1000
       a (i) = b(i) + c(i)
   enddo
!$OMP END DO [NOWAIT]
  ...
!$OMP PARALLEL
```

C/C++:

```
#pragma omp parallel [clauses]
{   ...
    #pragma omp for [clauses]
    {
      for (int i=0; i<1000; i++) {
          a[i] = b[i] + c[i];
      }
    }
...}
```

- **Threads share the work in loop parallelism.**
- **For example, using 4 threads under the "static" scheduling, in Fortran:**
  - **thread 1 has i=1-250**
  - **thread 2 has i=251-500, etc.**
- **Loop index i is private to each thread by default**

schedule Clause

- Static, #chunk: Loops are divided into chunks containing *#chunk* iterations. Assign chunks to #thrds in round-robin.
  - **Pre-determined and predictable by the programmer**
  - **Least work at runtime : scheduling done at compile-time**
- Dynamic, *#chunk*: Loops are divided into chunks containing *#chunk* iterations. Threads take chunks dynamically.
  - **Unpredictable, highly variable work per iteration**
  - **Most work at runtime : complex scheduling logic used at run-time**
- Guided, #chunk: Similar to dynamic, but loops are divided into progressively smaller chunks until the chunk size is 1.
- Auto: The compiler (or runtime system) decides what to use.
- Runtime: Use OMP_SCHEDULE environment variable to determine at run time.

Working with Loops

- Basic approach
  - Find compute intensive loops
  - Make the loop iterations independent ... So they can safely execute in any order without loop-carried dependencies
  - Place the appropriate OpenMP directive and test

```
int i, j, A[MAX];
 j = 5;
 for (i=0;i< MAX; i++) {
    j +=2;
    A[i] = big(j);
}
```

Working with Loops

- Basic approach
  - Find compute intensive loops
  - Make the loop iterations independent ... So they can safely execute in any order without loop-carried dependencies
  - Place the appropriate OpenMP directive and test

```
int i, j, A[MAX];
 j = 5;
 for (i=0;i< MAX; i++) {
    j +=2;
    A[i] = big(j);
}
```

Note: loop index "i" is private by default

```
int i,  A[MAX];
#pragma omp parallel for
 for (i=0;i< MAX; i++) {
    int j = 5 + 2*(i+1);
    A[i] = big(j);
}
```

Remove loop carried dependence

nowait Clause

- Barriers are really expensive. You need to understand when they are implied and how to skip them when its safe to do so.

```
double A[big], B[big], C[big];

#pragma omp parallel
{
        int id=omp_get_thread_num();
        A[id] = big_calc1(id);
#pragma omp barrier
#pragma omp for
        for(i=0;i<N;i++){C[i]=big_calc3(i,A);}
#pragma omp for nowait
        for(i=0;i<N;i++){ B[i]=big_calc2(C,  i); }
        A[id] = big_calc4(id);
}
```

implicit barrier at the end of a for worksharing construct

implicit barrier at the end of a parallel region

no implicit barrier due to nowait

Loop Parallelism: ordered and collapse Directives

FORTRAN example:

```
!$OMP DO ORDERED
    do i = 1, 1000
       a (i) = b(i) + c(i)
    enddo
!$OMP END DO
```

FORTRAN example:

```
!$OMP DO COLLAPSE (2)
    do i = 1, 1000
       do j = 1, 100
          a(i,j) = b(i,j) + c(i,j)
       enddo
    enddo
!$OMP END DO
```

- **ordered specifies the parallel loop to be executed in the order of the loop iterations.**
- **collapse *(n)* collapses the *n* nested loops into 1, then schedule work for each thread accordingly.**

Combined parallel worksharing Constructs

FORTRAN:

```
!$OMP PARALLEL DO
    do i = 1, 1000
       a (i) = b(i) + c(i)
    enddo
!$OMP PARALLEL END DO
```

C/C++:

```
#pragma omp parallel for
    for (int i=0; i<1000; i++) {
       a[i] = b[i] + c[i];
    }
```

FORTRAN example:

```
!$OMP PARALLEL SECTIONS
!$OMP SECTION
     do i = 1, 1000
     c (i) = a(i) + b(i)
     enddo
!$OMP SECTION
     do i = 1, 1000
        d(i) = a(i) * b(i)
     enddo
!$OMP PARALLEL END SECTIONS
```

FORTRAN only:

```
INTEGER N, M
PARAMETER (N=100)
REAL A(N,N), B(N,N), C(N,N), D(N,N)
!$OMP PARALLEL WORKSHARE
     C = A + B
     M = 1
     D= A * B
!$OMP PARALLEL END WORKSHARE
```

reduction Clause

C/C++ example: Serial

```
double ave=0.0, A[max]; int i;
for (i=0; i< max; i++) {
      ave + = A[i];
 }
 ave = ave/max;
```

C/C++ example: Parallel

```
double ave=0.0, A[max]; int i;
#pragma omp parallel for reduction (+:ave)
 for (i=0; i< max; i++) {
      ave + = A[i];
 }
 ave = ave/max;
```

- **Common accumulation pattern, with loop dependencies in serial code.**
- **Syntax: Reduction (operator : list)**
- **Reduces list of variables into one, using operator**
- **Reduced variables must be shared variables**
- **Allowed Operators:**
  - **arithmetic: + - *; math: max min;**
  - **bitwise: & | ^; logical: && ||**
- **A local copy of each list variable is made and initialized depending on the "op" (e.g. 0 for +, 1 for *)**
- **Each thread does its local accumulation first, then a global reduction is done at the end**

User Defined Reductions

- For mathematically associative and commutative operations
- **Declare the reduction operator**
  - Name, type, combiner function, initialization of local copies
  - Use special identifiers **omp_in** for value to be combined, **omp_out** for resulting combined value, **omp_priv** to initialize private copy

```
int my_mul(int a, int b) { return a * b; }

#pragma omp declare reduction(mul_id : int : omp_out *= omp_in) \
    initializer(omp_priv = 1)
```

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

```
#pragma omp parallel for reduction(mul_id : prod_par)
    for (i = 0; i < ARRAY_SIZE; i += 1) {
        prod_par = my_mul(prod_par, array[i]);
    }
```

Pi with reduction

```
/* file name: pi_omp_reduction.c */

...
int main()
{
   int i; double x, pi, sum = 0.0;
   step = 1.0/(double) num_steps;

#pragma omp parallel for private(x) reduction (+:sum)
   for (i=0; i<=num_steps; i++)
   {
     x=(i+0.5)*step;
     sum=sum+ 4.0/(1.0+x*x);
   }
   pi = step * sum;
   printf("pi=%f\n",pi);
   return 0;
}
```

- "reduction" is a more scalable and elegant solution than "critical".
- Only 2 extra lines of codes than the serial code!

Parallelizing doacross Loop

```
#pragma omp for ordered(2) collapse(2)
  for (i=1; i<N; i++)
  {
    for (j=1; j<M; j++)
    {
      A[i][j] = foo(i, j);
  #pragma omp ordered depend(sink: i-1,j)
                      depend(sink: i,j-1)
      B[i][j] = foo(B[i-1][j], B[i][j-1]);
  #pragma omp ordered depend(source)
      C[i][j] = func2(B[i][j]);
    }
  }
}
```

Threads wait here until B[i-1,j] and B[i,j-1] have been released

B[i,j] is ready to be released for other threads to use

- Help with the cross-iteration dependencies in the doacross loop nest
- ordered(2) defined 2 loops contribute to such dependency
- Wait point defined with “depend(sink)”
- Release with “depend(source)”
- Use “ordered” clause to ensure structured blocks are executed on lexical order

48

OpenMP Data Environment

- Most variables are shared by default
  - Fortran: common blocks, SAVE variables, module variables
  - C/C++: file scope variables, static variables
  - Both: dynamically allocated variables
- Some variables are private by default
  - Certain loop indices
  - Stack variables in subroutines or functions called from parallel regions
  - Automatic (local) variables within a statement block

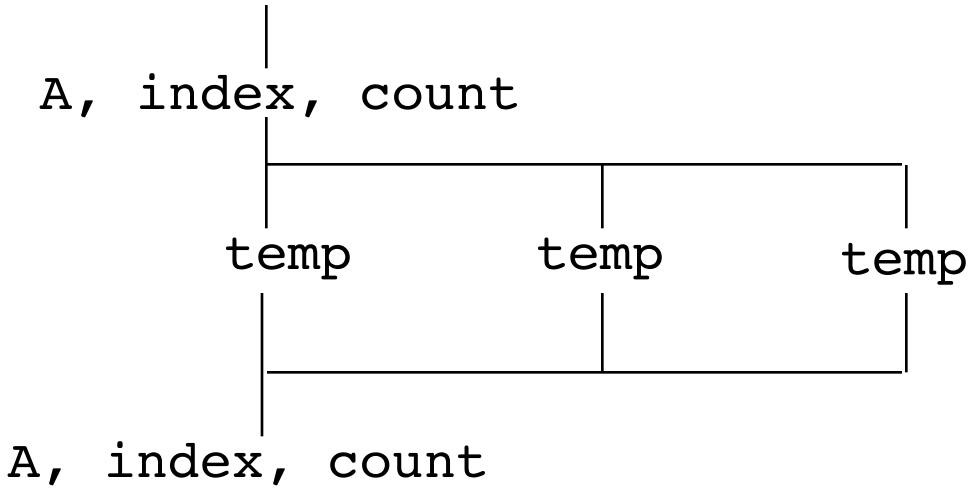
A Data Sharing Example

```
double A[10];
    int main() {
    int index[10];
   #pragma omp parallel
          work(index);
    printf("%d\n", index[0]);
  }
```

```
extern double A[10];
void work(int *index) {
  double temp[10];
  static int count;
  ...
}
```

A, index and count are shared by all threads.

temp is local to each thread



50

Data Sharing: private Clause

- private(var) creates a new local copy of var for each thread.
  - The value of the private copies is uninitialized
  - The storage of the private copy will be on the each thread's stack memory, and is unassociated with the original variable
  - The value of the original variable is unchanged after the region

```
void wrong() {
      int tmp = 0;
#pragma omp parallel for private(tmp)
      for (int j = 0; j < 1000; ++j)
            tmp += j;
      printf("%d\n", tmp);
}
```

tmp was not initialized

tmp is 0 here

When you need to reference the variable tmp that exists prior to the construct, we call it the **original variable**.

51

OMP_STACKSIZE

- OMP_STACKSIZE defines the private stack space each thread has.
- Default value is implementation dependent, and is usually quite small.
- Behavior is undefined if run out of space, mostly segmentation fault.
- To change, set OMP_STACKSIZE to n (B,K,M,G) bytes.
  - export OMP_STACKSIZE=32M (for bash/ksh)
  - setenv OMP_STACKSIZE 32M (for csh/tcsh)

Data Sharing: default Clause

- **default(none)**: Forces you to define the storage attributes for variables that appear inside the static extent of the construct … if you fail the compiler will complain. Good programming practice!
- You can put the default clause on parallel and parallel + workshare constructs.

The static extent is the code in the compilation unit that contains the construct.

```
#include <omp.h>
int main()
{
   int i, j=5;     double x=1.0, y=42.0;
   #pragma omp parallel for default(none) reduction(*:x)
   for (i=0;i<N;i++){
      for(j=0; j<3; j++)
         x+= foobar(i, j, y);
   }
   printf(" x is %f\n",(float)x);
}
```

The compiler would complain about j and y, which is important since you don't want j to be shared

53

Data Sharing: firstprivate Clause

FORTRAN Example:

```
PROGRAM MAIN
      USE OMP_LIB
      INTEGER I
      I = 1
!$OMP  PARALLEL FIRSTPRIVATE(I) &
!$OMP PRIVATE(tid)
      I = I + 2    ! I=3
      tid = OMP_GET_THREAD_NUM()
      if (tid ==1) PRINT *, I    ! I=3
!$OMP END PARALLEL
     PRINT *, I    ! I=1
END PROGRAM
```

- **Declares variables in the list private**
- **Initializes the variables in the list with the value when they first enter the construct.**

Data Sharing: threadprivate and copyin Clauses

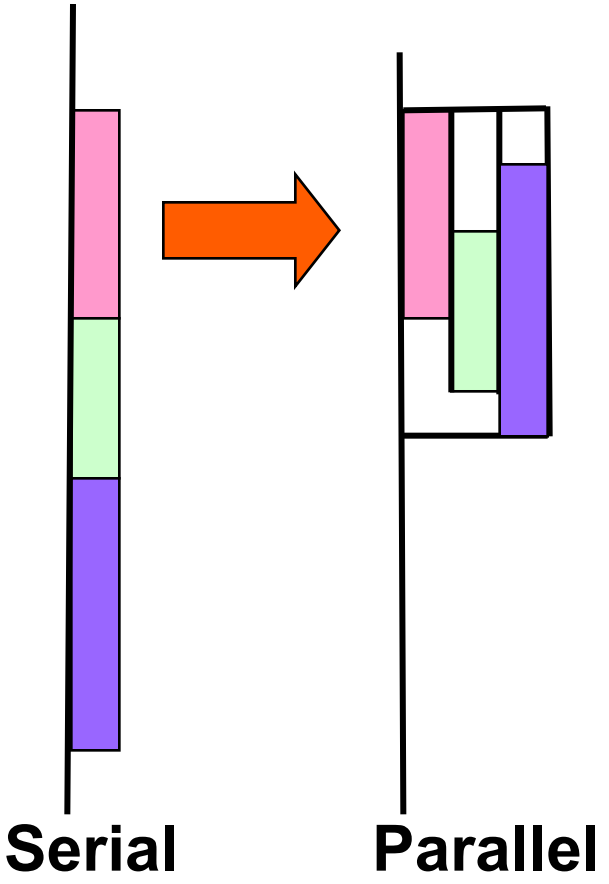
FORTRAN Example:

```
PROGRAM main use
 OMP_LIB
      INTEGER tid, K
      COMMON /T/K
!$OMP THREADPRIVATE(/T/) K
      = 1

!$OMP   PARALLEL PRIVATE(tid) COPYIN(/T/)
      PRINT *, "thread ", tid, ” ,K= ", K
      tid = omp_get_thread_num()
      K = tid + K
      PRINT *, "thread ", tid, “ ,K= ", K
!$OMP END PARALLEL

!$OMP   PARALLEL PRIVATE(tid)
      tid = omp_get_thread_num()
      PRINT *, "thread ", tid, ” ,K= ", K
!$OMP END PARALLEL END
 PROGRAM main
```

- **Threadprivate makes global data private to a thread and persistent between parallel regions**
  - **Fortran: COMMON blocks**
  - **C: File scope and static variables**
- **The copyin clause: copies the threadprivate variables from master thread to each local thread.**

What are tasks?

- Task construct: a structured block of code + a data environment
- Inside a parallel region, a thread encountering a task construct will package up the code block and its data for execution
- The task is executed immediately, or deferred for later execution.
- Tasks can be nested: i.e. a task may itself generate tasks.



A common Pattern is to have one thread create the tasks while the other threads wait at a barrier and execute the tasks

Tasking: task and taskwait Directives

Serial:

```
int fib (int n)
{
   int x, y;
   if (n < 2) return n;
   x = fib (n – 1);
   y = fib (n – 2);
   return x+y;
}
```

OpenMP:

```
int fib (int n) {
   int x,y;
   if (n < 2) return n;
#pragma omp task shared (x)
   x = fib (n – 1);
#pragma omp task shared (y)
   y = fib (n – 2);
#pragma omp taskwait
   return x+y;
}
```

- Flexible and powerful. Useful for situations when loops counts are not known at compile time.
- The task directive defines an explicit task. Threads share work from all tasks in the task pool.
- The taskwait directive makes sure all child tasks created for the current task finish.
- Helps to improve load balance.

Tasks example: Racey Cars

```
#include <stdio.h>
#include <omp.h>
int main()
{  printf("I think");
   #pragma omp parallel
   {
      #pragma omp single
      {
         #pragma omp task
            printf(" car");
         #pragma omp task
            printf(" race");
      }
   }
   printf("s");
   printf(" are fun!\n");
}
```

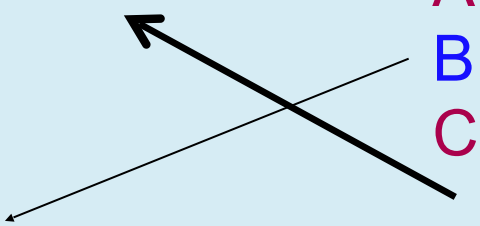
The output can sometimes be:
I think car races are fun!

and sometimes be:
I think race cars are fun!

OpenMP Tasks: Data Scoping Defaults

- The behavior you want for tasks is usually **firstprivate**, because the task may not be executed until later (and variables may have gone out of scope)
  - **Variables that are private when the task construct is encountered are firstprivate by default**
- Variables that are shared in all constructs starting from the innermost enclosing parallel construct are shared by default

```
#pragma omp parallel shared(A) private(B)
{
    ...
#pragma omp task
    {
        int C;
        compute(A, B, C);
    }
}
```



A is shared
B is firstprivate
C is private

Parallel Linked List Traversal

Serial code:

```
p = listhead ;
while (p) {
  process(p);
  p=next(p) ;
}
```

- Classic linked list traversal
- Do some work on each item in the list
- Assume that items can be processed independently
- Cannot use an OpenMP worksharing loop directive

```
#pragma omp parallel
{
  #pragma omp single
   {
    p = listhead ;
    while (p) {
       #pragma omp task firstprivate(p)
              {
                process (p);
              }
       p=next (p) ;
     }
   }
}
```

Only one thread packages tasks

Makes a copy of `p` when the task is packaged

Pi with Tasks

```
#include <omp.h>
static long num_steps = 100000000;
#define MIN_BLK  10000000
double pi_comp(int Nstart,int Nfinish,double step)
{   int i,iblk;
   double x, sum = 0.0,sum1, sum2;
   if (Nfinish-Nstart < MIN_BLK){
      for (i=Nstart;i< Nfinish; i++){
         x = (i+0.5)*step;
         sum = sum + 4.0/(1.0+x*x);
      }
   }
   else{
      iblk = Nfinish-Nstart;
      #pragma omp task shared(sum1)
           sum1 = pi_comp(Nstart, Nfinish-iblk/2,step);
      #pragma omp task shared(sum2)
           sum2 = pi_comp(Nfinish-iblk/2, Nfinish,  step);
      #pragma omp taskwait
         sum = sum1 + sum2;
   }return sum;
}
```

```
int main ()
{
  int i;
  double step, pi, sum;
   step = 1.0/(double) num_steps;
   #pragma omp parallel
   {
      #pragma omp single
          sum =
pi_comp(0,num_steps,step);
    }
     pi = step * sum;
}
```

Recursive divide-and-conquer algorithm

Pi OpenMP results

threads	1st SPMD	1st SPMD padded	SPMD critical	PI Loop and reduction	Pi tasks
1	1.86	1.86	1.87	1.91	1.67
2	1.03	1.01	1.00	1.02	1.00
3	1.08	0.69	0.68	0.80	0.76
4	0.97	0.53	0.53	0.68	0.52

- 1st SPMD uses array for sum ,has false sharing, hurt speedup.
- 1st SPMD uses padded array for sum, avoided false sharing, good speedup.
- SPMD critical uses scalar for sum, no false sharing, good speedup.
- Pi Loop and reduction uses scalar for partial sum, then reduction, relative good speedup, with some loop overhead.
- Pi tasks performs surprisingly well in this test. It is not reproducible on the HPC systems we tested with various compilers. Normally, do no use tasks if an algorithm is already well supported by OpenMP, such as standard do/for loop.

*Intel compiler (icpc) with default optimization level (O2) on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

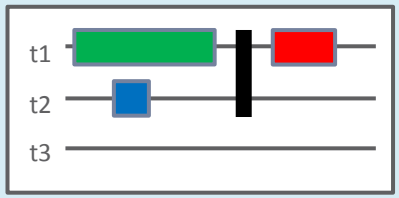
taskgroup Directive

- OpenMP 4.0 extends the tasking support.
- The taskgroup directive waits for all child tasks and any of their descendant tasks descendant tasks to complete as compared to taskwait which only waits for direct children.

Task synchronization with dependencies

OpenMP 3.1

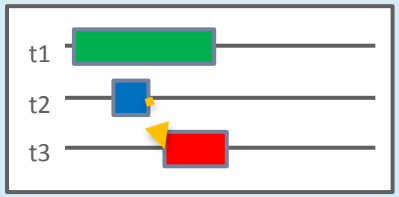
```
int x = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task
  std::cout << x << std::endl;

  #pragma omp task
  long_running_task();
  #pragma omp taskwait
  #pragma omp task
  x++;
}
```



Red task could not start until both green and blue task completes

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
  long_running_task();
  #pragma omp task depend(inout: x)

  x++;
}
```



Red task could start as soon as blue task completes

64

Taskloop: split worksharing-loop into tasks

```
Original code:
for (i = 0; i<SIZE; i+=1) {
   A[i]=A[i]*B[i]*S;
}
```

```
Non-blocking parallel worksharing:

#pragma omp parallel for
   shared(S,A,B)
   for (i = 0; i<SIZE; i+=1) {
      A[i]=A[i]*B[i]*S;
}
```

```
taskloop:

#pragma omp taskloop
for (i = 0; i<SIZE; i+=1) {
   A[i]=A[i]*B[i]*S;
}
```

- Loop iterations are turned into tasks that execute within a taskgroup
- Allows tasks to be executed in any order
- Provides dynamic load balance

65

taskloop "grainsize" clause simplifies blocking

```
original code

for (i = 0; i<SIZE; i+=1) {
   A[i]=A[i]*B[i]*S;
}
```

```
blocking tasking
for (i = 0; i<SIZE; i+=TS) {
   UB = SIZE < (i+TS) ? SIZE :
   i+TS;
   #pragma omp task private(ii) \
      firstprivate(i,UB)
   shared(S,A,B)
   for (ii=i; ii<UB; ii++) {
      A[ii]=A[ii]*B[ii]*S;
   }
}
```

Taskloop:

```
#pragma omp taskloop grainsize(TS)
for (i = 0; i<SIZE; i+=1) {
  A[i]=A[i]*B[i]*S;
}
```

- Taskloop provides same blocking as done manually, which is error-prone
- Taskloop improves programmability

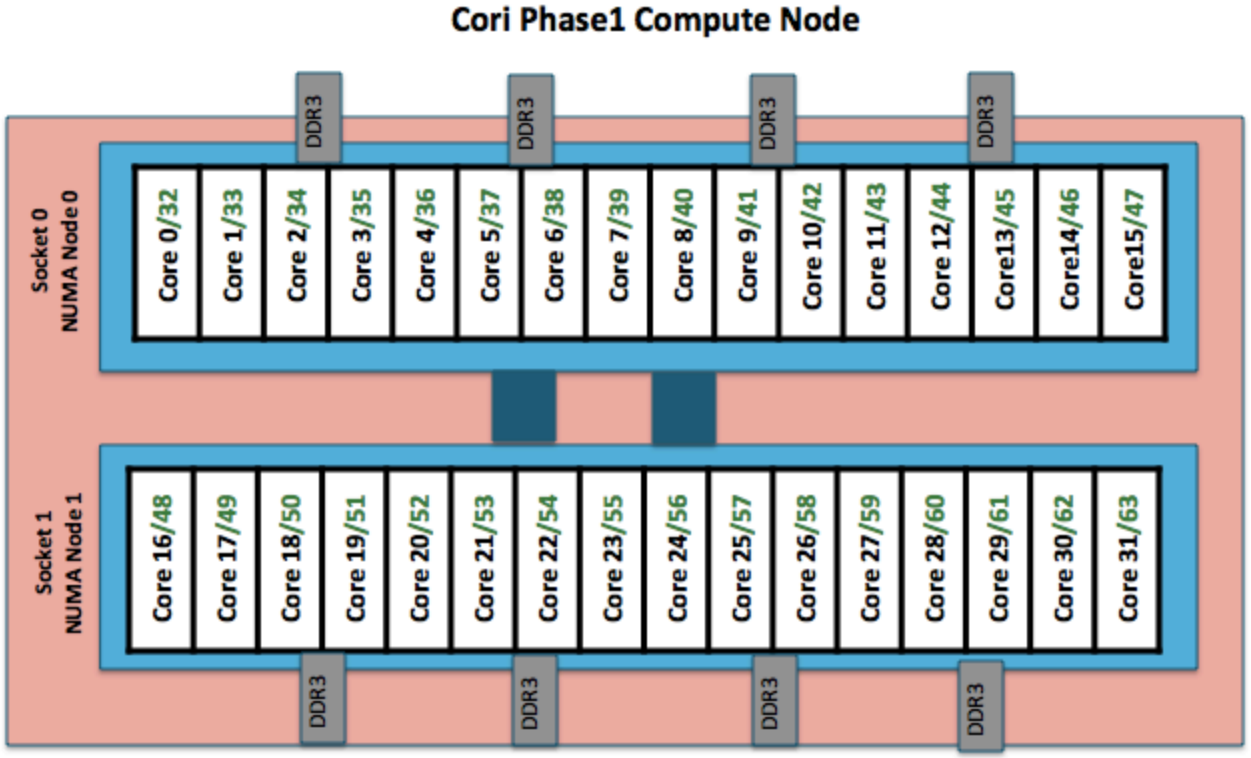
Thread Affinity and Data Locality

- Affinity
  - Process Affinity: bind processes (MPI tasks, etc.) to CPUs
  - Thread Affinity: further binding threads to CPUs that are allocated to their parent process

- Data Locality
  - Memory Locality: allocate memory as close as possible to the core on which the task that requested the memory is running
  - Cache Locality: use data in cache as much as possible

- Correct process, thread and memory affinity is the basis for getting optimal performance.

Example Compute Nodes (Intel Haswell*)



numactl -H: provides NUMA info of the CPUs

- **An Intel Haswell node has 32 cores (64 CPUs), 128 MB DDR memory.**
- **2 NUMA domains per node, 16 cores per NUMA domain. 2 hardware threads (CPUs) per core.**
- **Memory bandwidth is non-homogeneous among NUMA domains.**
  - **CPUs 0-15, 32-47 are closer to memory in NUMA domain 0, farther to memory in NUMA domain 1.**
  - **CPUs 16-31, 48-64 are closer to memory in NUMA domain 1, farther to memory in NUMA domain 0.**

*Haswell: 16-core **Intel® Xeon™ Processor E5-2698 v3 at 2.3 GHz**

Thread Affinity Control

- OpenMP 4.0 added **OMP_PLACES** environment variable to control thread allocation
  - Can be threads, cores, sockets, or a list with explicit CPU ids.
- **OMP_PROC_BIND** controls thread affinity within and between OpenMP places
  - OpenMP 3.1 only allows TRUE or FALSE.
  - OpenMP 4.0 still allows the above. Added options: close, spread, master.

Runtime Environment Variable: OMP_PROC_BIND

- Use 4 cores total, 2 hyperthreads per core, and OMP_NUM_THREADS=4 as example
- none: no affinity setting.
- close: Bind threads as close to each other as possible

Node	Core 0		Core 1		Core 2		Core 3	
	HT1	HT2	HT1	HT2	HT1	HT2	HT1	HT2
Thread	0	1	2	3				

- spread: Bind threads as far apart as possible.

Node	Core 0		Core 1		Core 2		Core 3	
	HT1	HT2	HT1	HT2	HT1	HT2	HT1	HT2
Thread	0		1		2		3	

- master: bind threads to the same place as the master thread

Memory Affinity: "First Touch" Memory

Step 1.1 Initialization by master thread only

```
for (j=0; j<VectorSize; j++) {
a[j] = 1.0; b[j] = 2.0; c[j] = 0.0;}
```

Step 1.2 Initialization by all threads

```
#pragma omp parallel for
for (j=0; j<VectorSize; j++) {
a[j] = 1.0; b[j] = 2.0; c[j] = 0.0;}
```

Step 2 Compute

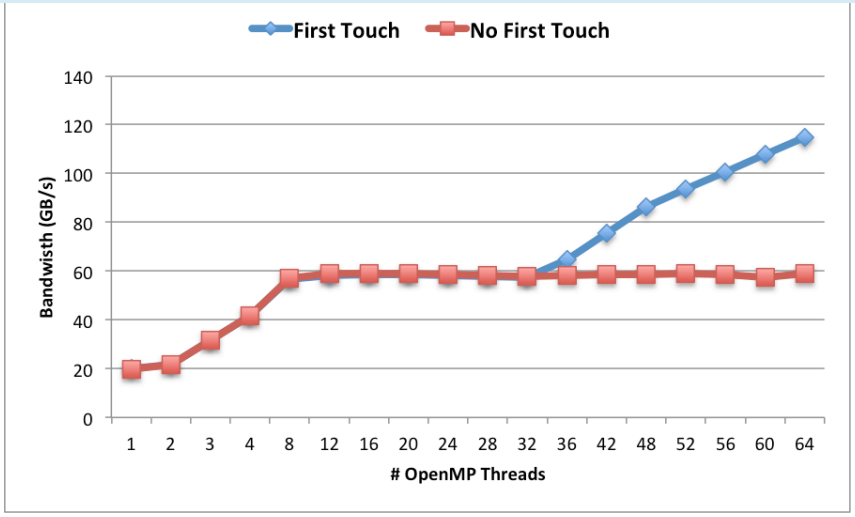
```
#pragma omp parallel for
for (j=0; j<VectorSize; j++) {
a[j]=b[j]+d*c[j];}
```

OMP_PROC_BIND=close

- Memory affinity is not defined when memory was allocated, instead it will be defined at initialization.
- Memory will be local to the thread which initializes it. This is called **first touch** policy.

Hard to do "perfect touch" for real applications.

General recommendation is to use number of threads fewer than number of CPUs per NUMA domain.



OMP_PROC_BIND Choices for STREAM

OMP_NUM_THREADS=32
OMP_PLACES=threads

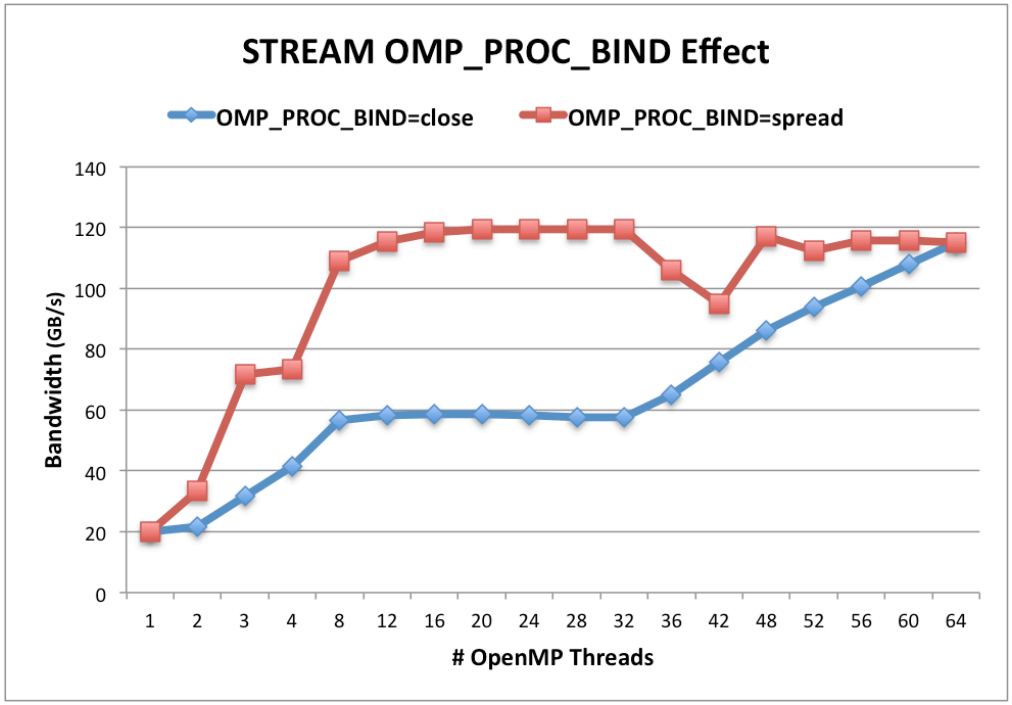
OMP_PROC_BIND=close

Threads 0 to 31 bind to CPUs 0,32,1,33,2,34,…15,47. All threads are in the first socket. The second socket is idle. Not optimal.

OMP_PROC_BIND=spread

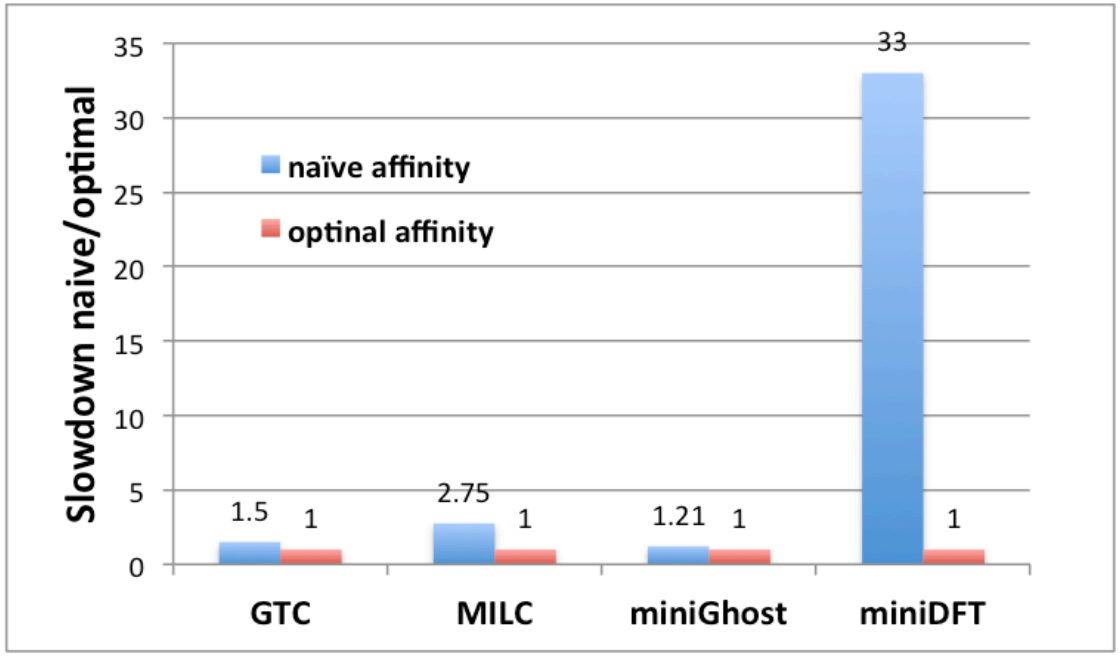
Threads 0 to 31 bind to CPUs 0,1,2,… to 31. Both sockets and memory are used to maximize memory bandwidth.

Blue: OMP_PROC_BIND=close
Red: OMP_PROC_BIND=spread
Both with First Touch



Naïve vs. optimal affinity

Application Benchmark Performance on Cori



- Use the display affinity feature in OpenMP 5.0, available in many major compilers already:
  - Two runtime environment variables
    - OMP_DISPLAY_AFFINITY
    - OMP_DISPLAY_AFFINITY_FORMAT
  - Runtime APIs to get/set the thread affinity info

What Does the User Have to Do?

- Starting point is most often MPI or sequential program code
- Application developer must decide how the work can be divided up among multiple threads
  - Identify parallelism and needed synchronization
  - Getting this right is the **user's responsibility**!
  - Insert OpenMP constructs that represent the strategy
- Getting good performance requires an understanding of implications of chosen strategy
  - Translation introduces overheads
  - Data access pattern might affect performance
- **Sometimes, non-trivial rewriting of code is needed to accomplish desired results**

User makes strategic decisions; compiler figures out details

Fine Grain and Coarse Grain Models

```
Program fine_grain
!$OMP PARALLEL DO
    do i=1,n
        … computation
    enddo
!$OMP END PARALLEL DO

  … some serial computation …

!$OMP PARALLEL DO
    do i=1,n
        … computation
    enddo
!$OMP END PARALLEL DO
end
```

- Program is single threaded except when actively using multiple threads, such as loop processing,
- Pro: Easier to adapt to MPI program.
- Con: thread overhead, serial section becomes bottleneck.

```
Program coarse_grain
!$OMP PARALLEL
!$OMP DO
    do i=1,n
        … computation
    enddo
!$OMP END DO

!$OMP DO
    do i=1,n
        … computation
    enddo
!$OMP END DO
!$OMP END PARALLEL
 end
```

- Majority of program run within an OMP parallel region.
- Pro: low overhead of thread creation, consistent thread affinity.
- Con: harder to code, prone to race condition.

Why not perfect speedup with OpenMP?

Jacobi OpenMP	Execution Time (sec)	Speedup
1 thread	121	1
2 threads	63	1.92
4 threads	36	3.36

- Possible causes
  - Serial code sections not parallelized
  - Thread creation and synchronization overhead
  - Memory bandwidth
  - Memory access with cache coherence
  - Possible false sharing
  - Load balancing
  - Not enough work for each thread

Best practices for OpenMP

- Use profiling tools to find hotspots. Add OpenMP and check correctness incrementally
- Parallelize as much as possible
  - Amdahl's law: **If 1/s of the program is sequential, then you cannot ever get a speedup better than s**
- Choose between fine grain or coarse grain parallelism implementation
- Reduce number of OpenMP parallel regions to reduce overhead costs
- Parallelize outer loop and consider loop collapse, loop fusion or loop permutation to give all threads enough work, and to optimize thread cache locality. Use NOWAIT clause if possible
- Minimize shared variables, minimize serial/critical/barrier sections
- Pay attention to load imbalance. If needed, try dynamic scheduling or implement own load balance scheme
- Consider OpenMP Tasks
- Consider nested OpenMP
- Consider OpenMP SIMD for better vectorization
- Be aware of NUMA domains
- Look out for false sharing

Agenda

- Part 1 (ca. 65 min):
  - **Background**
  - **Parallel Regions**
  - **Working with Loops**
  - **Data Environment**
  - **Tasks**
  - **Thread Affinity**

Presenter: **Helen He**

- Part 2 (ca. 45 min)
  - **Using GPUs**
  - **SIMD Features**
  - **OpenMP 5.0 Topics**
  - **Wrap Up / More Info**

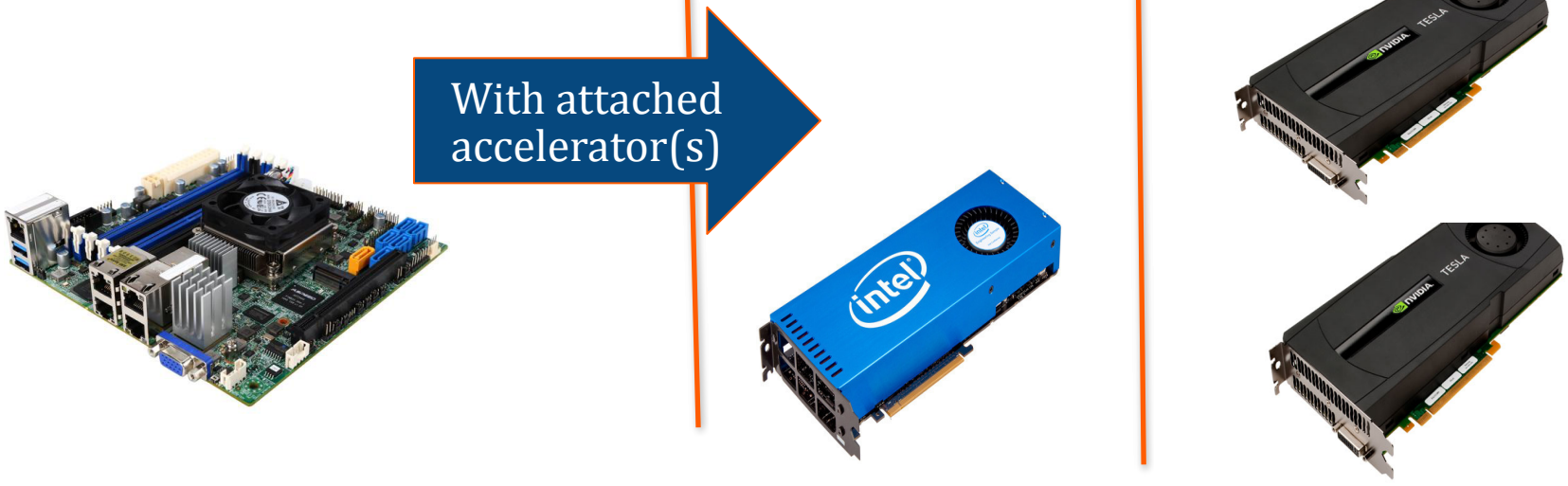
Presenter: **Oscar Hernandez**

What's new in OpenMP 4.0/4.5

- Directives
  - Target regions (to support accelerators)
    - structure and unstructured target data regions
    - Asynchronous execution (nowait) and data dependences (depend)
  - SIMD (to support SIMD parallelism)
  - New tasking features
    - taskloops, groups, dep, priorities
    - Cancellation
  - Thread affinity
    - Per parallel region (including nested parallelism)
  - Do across
    - Ordered (do across)
- Runtime APIs
  - Target regions, data mapping APIs
- Environment Variables
  - Affinity:
    - OpenMP Places (OMP_PLACES)
      - Hardware abstraction
    - Thread bindings (OMP_PROC_BIND)
      - Controls the mapping of threads to places
  - Target
    - Default accelerator type

OpenMP 4.0/4.5 – Accelerator model

- OpenMP 4.0/4.5 supports heterogeneous systems (accelerators/devices)
- Accelerator model
  - One host device and
  - One or more target devices



Host Device
(CPU Multicore)

Xeon Phi(s) –
(Accelerator and self-hosted)

GPU(s)

OpenMP Target

- Device:
  - An implementation-defined (logical) execution unit (or accelerator)
- Device data environment
  - Storage associated with the device
- The execution model is host-centric (or initial device)
  - Host creates/destroys data environment on device(s)
  - Host maps data to the device(s) data environment
  - Host offloads OpenMP target regions to target device(s)
  - Host updates the data between the host and device(s)

OpenMP 4.5 Device Constructs

- Execute code on a target device
  - **omp target** *[clause[[,] clause],…]*
    structured-block
  - **omp declare target**
    [function-definitions-or-declarations]
- Manage the device data environment
  - **map** ([map-type:] list)
    map-type := alloc | *tofrom* | *to* | *from*
    | *release* | *delete*
  - **omp target data** [clause[[,] clause], …]
    structured-block
  - **omp target enter/exit data** [clause[[,] clause], …]
  - **omp target update** [clause[[,] clause], …]
  - **omp declare target**
    [variable-definitions-or-declarations]

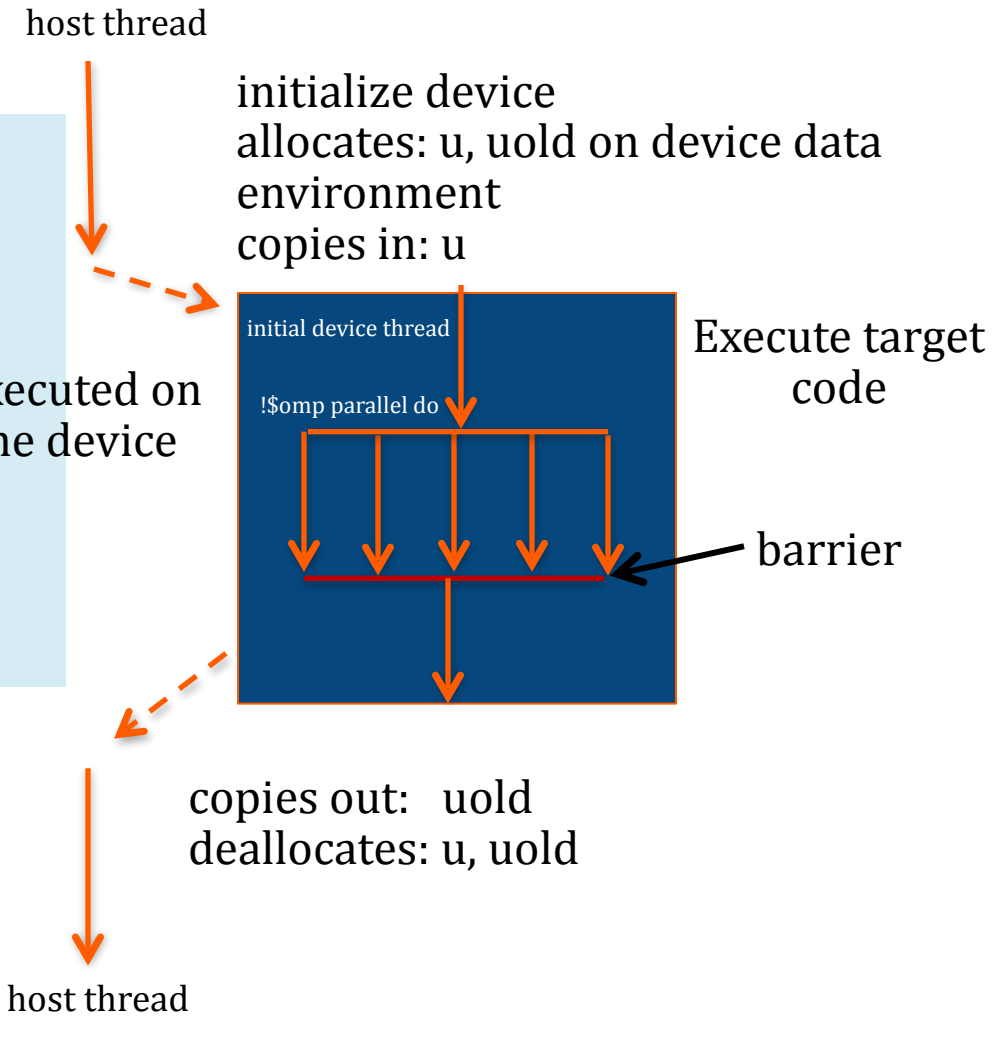
- Parallelism & workshare for devices
  - **omp teams** *[clause[[,] clause],…]*
    structured-block
  - **omp distribute** *[clause[[,] clause],…]*
    for-loops
- Device Runtime Support
  - `void omp_set_default_device(int dev_num)`
  - `int omp_get_default_device(void)`
  - `int omp_get _num_devices(void)`
  - `int omp_get_team_num(void)`
  - `int omp_is_initial_device(void)`
- Environment variables
  - OMP_DEFAULT_DEVICE
  - OMP_THREAD_LIMIT

OpenMP Target

OpenMP for accelerators

```
…
!$omp target map(to:u) map(from:uold)

!$omp parallel do collapse(2)
    do j=1,m
        do i=1,n
            uold(i,j) = u(i,j)
        enddo
    enddo

!$omp end target
…
```

Executed on the device



Use target construct to:

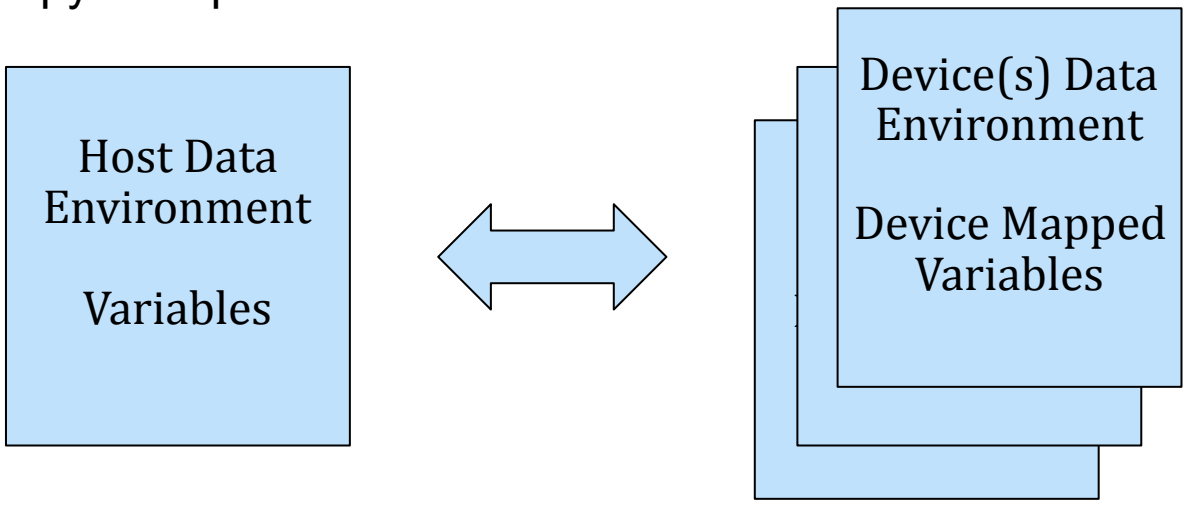
- Transfer control from the host to the target device
- Map variables to/from the device data environment

Host thread waits until target region completes

- Use nowait for asynchronous execution

OpenMP Target and Data Regions

- The map clauses determine how an original (initial device) variable in a data environment is mapped to a corresponding variable in a device data environment
  - Mapped variable:
    - An original variable in a (host) data environment has a corresponding variable in a device data environment
  - Mapped type:
    - A type that is amenable for mapped variables
    - Bitwise copy-able plus additional restrictions



OpenMP Target data environment

- Data environment on a target directive
  - `#pragma omp target map(tofrom: x)`

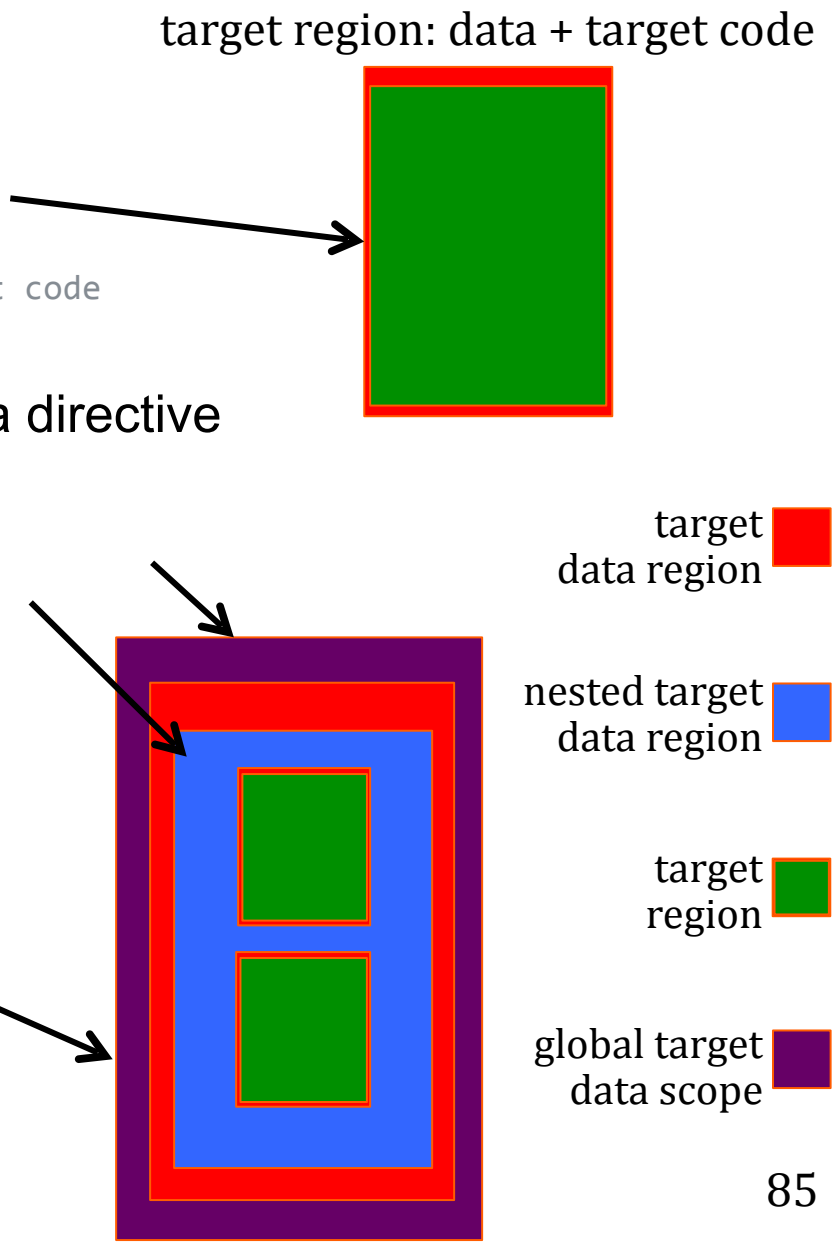
```
{
  // scope of the data environment and target code
}
```

- Data environment scoped via target data directive
  - `#pragma omp target data map(tofrom: x)`

```
{
  // scope of the data environment only
}
```

- Global scope data environment
  - `#pragma omp declare target to(x)`

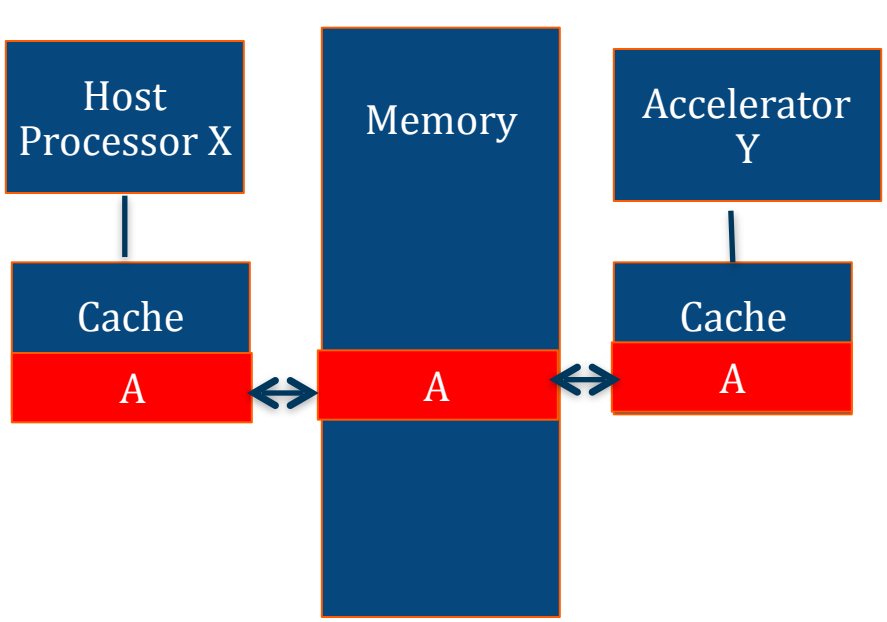
- Unstructured data regions
  - `#pragma omp enter/exit data map(x)`



OpenMP Target data regions – implementation

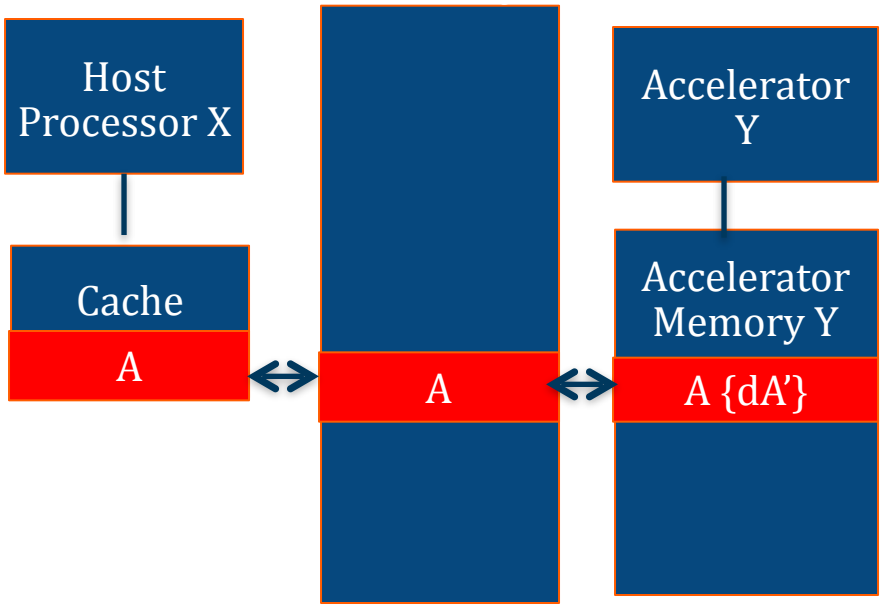
```
#pragma omp target data map(tofrom: A) {…}
```

Shared memory system implementation



* One copy on multiple cache lines that need to synchronize

Discrete memory systems implementation



* Two copies that need to synchronize:
A – host , A – device (dA)

OpenMP Map-types to device data regions

- Map-types:
  - alloc - allocates data on the device
  - to - allocates data and moves data to the device
  - from – allocates data and moves data from the device
    - for **omp target exit data** only moves the data from the device
  - tofrom – allocates data and moves data to and from the device
  - delete – deletes the data from the device and sets the reference count to 0
  - release – decrements the reference count of a variable
- Map-type Modifiers
  - always

map-type	target	target data	target enter data	target exit data	target declare
alloc	X	X	X		X
to	X	X	X		X
from	X	X		X	
tofrom	X	X			
delete				X	
release				X	

Reference Counts

- Reference count for each mapped variable
- Incremented by 1 for each of the following constructs
  - Entry of target / target data
  - Target enter data
- Decremented by 1 for each of the following constructs
  - On exit of a target / target data
  - Target exit data
- Exceptions
  - target declare data environment (global) infinite ref counts
  - target exit data map-type is delete – ref counts to zero
    - `#pragma omp target exit data map(delete: A)`

Example: OpenMP data regions

```
double A[N], B[N], C[N], D[N];
#pragma omp target data map(alloc: C)
{
    #pragma omp target data map(to: A, B) map(from: C)
    {
        /* device code */
    }
    #pragma omp target map(from: D) map(to: C)
    {
        /* device code */
    }
}
```

C is already in device scope

ref_count(C)=1

Thus inner map clauses of C are ignored

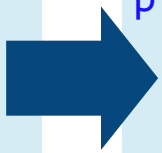
*OpenMP also provide mapping support for array sections A[0:N]

Default Mappings for device data regions

- Arrays: map(tofrom: A)
  - Array are mapped in their entirety
- Scalars: firstprivate(i)
- Pointers: map(tofrom: p[0:0])
  - Default is zero-length pointers
  - Pointers initialized to NULL

Example with default mappings:

```
int A[N], B[N], *p, len=N;
if(x) p = &A;
else p = &B;

#pragma omp target
{
    // A, B, p, len are mapped by default

    for(i = 0; i < len; i++)
        A[i] = B[i] + p[i];
}
```



Example with explicit mappings:

```
int A[N], B[N], *p, len=N;

if(x) p = &A;
else p = &B;

#pragma omp target data map(tofrom:
A, B)
{
   /* A and B are mapped tofrom */

   #pragma omp target map(to:
p[0:0]) firstprivate(len)
   {
   /* len is mapped firstprivate
       with value N
       p is mapped as a zero-lengh
       pointer where:
       p == &dA    // if (x) == true
       p == NULL  // else  */
     …
   }
}
```

OpenMP Target update construct

- Can be used to specify data transfers between host and devices
- #pragma omp target update [clause[[,] clause],…]
- clauses:
  - device(scalar-integer-expression)
  - to (list)
  - from (list)
  - if (scalar-expr)
  - nowait
  - depend (dependence-type:list)

Example:

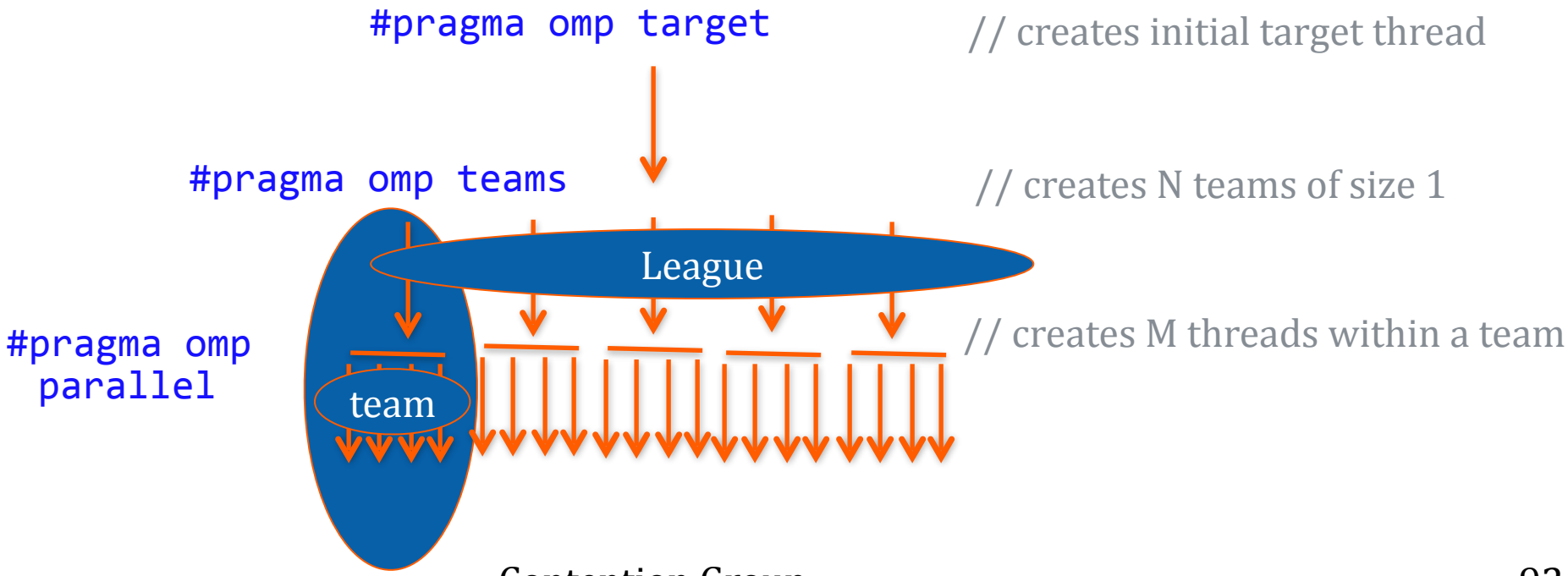
```
/* device do some work */
#pragma omp target update from(A)
 …

/* host do some work */
#pragma omp target update to(A)
```

Updates A device->host

Updates A host->device

League and teams of threads

- League
  - Set of thread teams created by a teams construct
- Contention group
  - Threads of a team in a league and their descendant threads
  - Threads can synchronize in the same contention group



OpenMP Teams construct

- The **teams** construct creates a league of thread teams
  - The master threads of all teams executes the team region
  - The number of teams is specified by the **num_teams** clause
  - Each team executes with **thread_limit** threads
  - Threads in different teams cannot synchronize with each other
  - Must be perfectly nested in a target construct
    - No statements or directives between teams and target constructs
  - Only special openmp constructs can be nested inside a teams construct
    - distribute
    - parallel
    - parallel for
    - parallel sections

OpenMP Distribute construct

- Work-sharing construct for target and teams regions
  - Distribute the iterations of a loop across the master threads of the teams executing the region
  - No implicit barrier at the end of the construct
- dist_schedule(kind[, chunk_size])
  - kind must be static scheduling
  - Chunks are distribute in round-robin fashion with chunk_size
  - Each team receives at least one evenly distributed chunk (if no chunk_size is specified)

```
#pragma omp target map(tofrom: A)
#pragma omp teams
#pragma omp distribute
    for (i=0; i< N; i ++)
        A[i] = …
```

Putting it together …

```
#define N 128
double x[N*N];
int i, j, k;
for (k=0; k<N*N; ++k) x[k] = k;

#pragma omp target
#pragma omp teams distribute
for (i=0; i<N; ++i) {
#pragma omp parallel for
  for (j=0; j<N; ++j) {
    x[j+N*i] *= 2.0;
  }
}
```

- The **target** construct offloads the enclosed code to the accelerator
- The **teams** construct creates a league of teams
- The **distribute** construct distributes the outer loop iterations between the league of teams
- The **parallel for** combined construct creates a thread team for each team and distributes the inner loop iterations to threads

Writing Portable Device code

```
#pragma omp target map(tofrom: A)
#pragma omp teams distribute parallel for simd collapse(3)
// combined directive
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      for (k=0; k<N; k++)
        A[i][j][k] = …
```

- Use OpenMP 4 "Accelerator Model" to target multiple architectures: GPUs, Intel Xeon Phi, and multicore CPUs, etc.
- Make your OpenMP adaptable or using defaults for:
  - # of teams,
  - dist_schedule,
  - thread_limit #,
  - # of threads in parallel regions,
  - parallel for loop schedules,
  - SIMD length

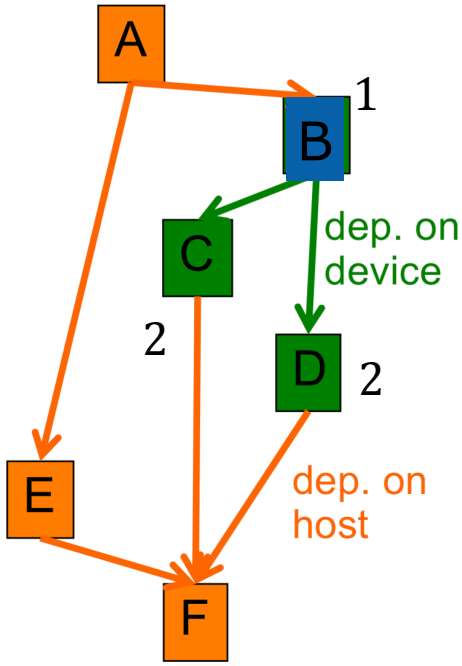
Example:

- Xeon Phi implementation may chose num_teams(1), thread_limit(1) and simdlen(V)
- GPUs implementation may chose num_teams(N), thread_limit(M) and simdlen(V)

Task Parallelism with Target

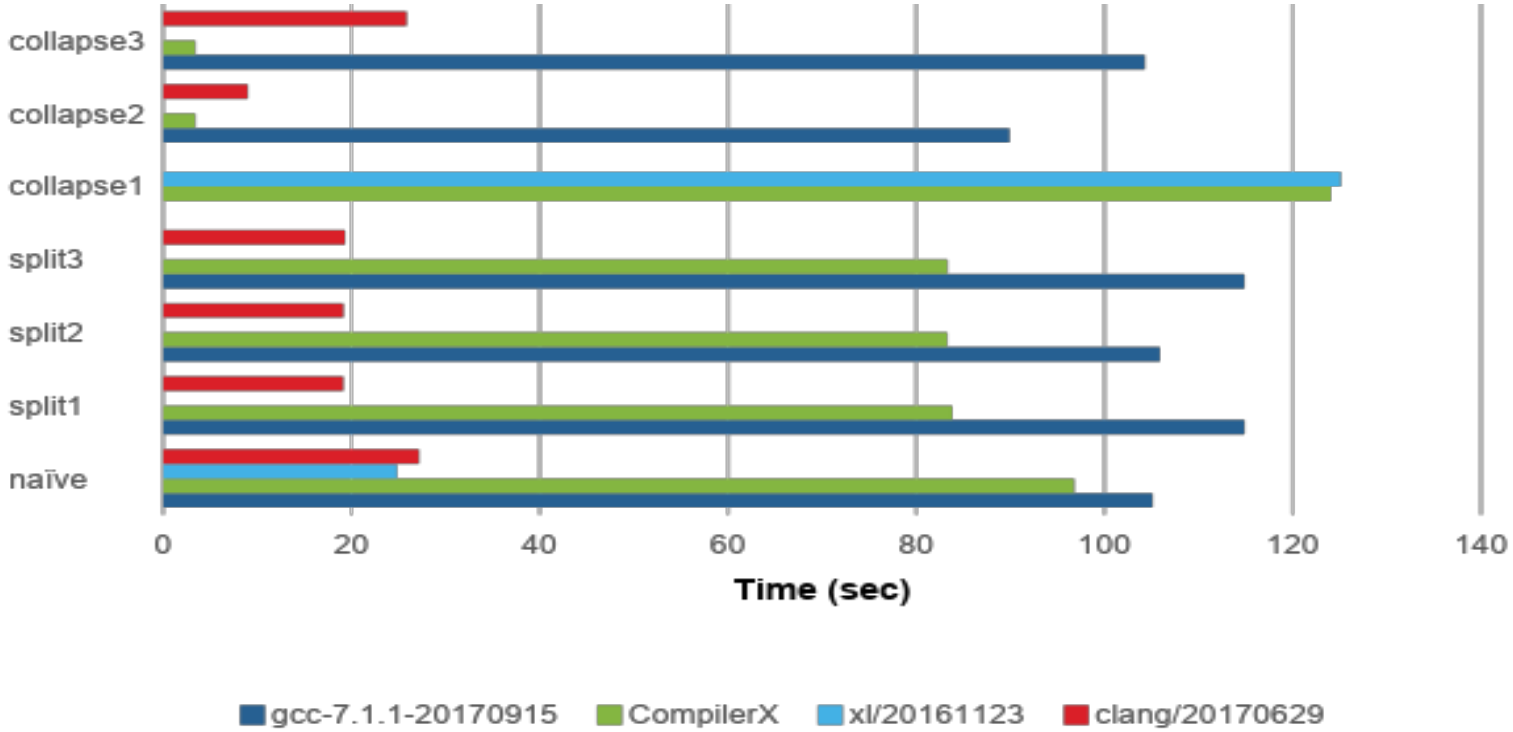
- OpenMP 4.5 tasking model includes the target model
  - A target region has an implicit task

```
#pragma omp task depend(out: A) {Code A}
#pragma omp target depend(in: A)  nowait
                          depend(out: B) device(1) {Code B}
#pragma omp target depend(in: B) nowait
                           depend(out: C) device(2) {Code C}
#pragma omp target depend(in: B) nowait
                            depend(out: D) device(2) {Code D}
#pragma omp task depend(in: A)
                          depend(out: E)  {Code E}
#pragma omp task depend(in: C,D,E) {Code F}
```



MPI + OpenMP with GPU offload– Lessons Learned

- Matrix Multiplication: MPI for intranode communications and OpenMP to offload computation to GPUs
  - Application developers must pay **extreme** attention on tuning or performance won't be there.
  - Compilers/runtimes need to help for performance portability across platforms.



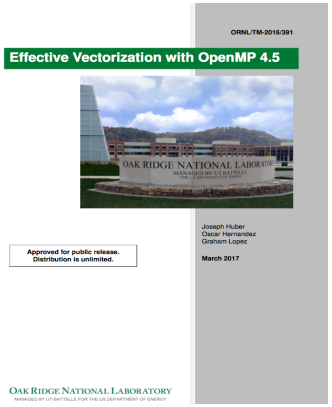
Summitdev:: Power8, NVIDIA GP100, NVLINK, IB EDR

OpenMP SIMD directive

- Hints the compiler to SIMD'ed code
  - Can be used on the host or inside target regions
    - Compilers have different strategies to lower SIMD on different architectures.
  - E.g. generate vector instructions, co-operating threads (in GPU), etc
- `#pragma omp simd [clause[ [, ]clause] ...]`

  for-loops
  - clauses:
    - safelen(length)
    - simdlen(length)
    - linear(list[: linear-step])
    - aligned(list[: alignment])
    - private(list[: alignment])
    - lastprivate(list)
    - reduction(reduction-identified :list)
    - Collapse(n)

How OpenMP SIMD works for Vectorization



http://info.ornl.gov/sites/publications/files/Pub69214.pdf

OpenMP SIMD directive

- Align data on specific byte boundaries; directive based approach with OpenMP directive:
  - Portable solution: `!$OMP SIMD ALIGNED (…)`
    - Tells the compiler that the arrays are aligned (shown to improve 8% of performance)
    - Asserts that there are no dependencies
    - Requires to use PRIVATE or REDUCTION clauses to ensure correctness
    - Forces the compiler to vectorize, whether or not it thinks if it is profitable or not
  - As compared to: `!DIR$ VECTOR ALIGNED`
    - Tells the compiler that the arrays are aligned
    - Intel compiler specific, not portable
- `!$OMP SIMD ALIGNED` is independent of vendor, however it can be overly intrusive in code

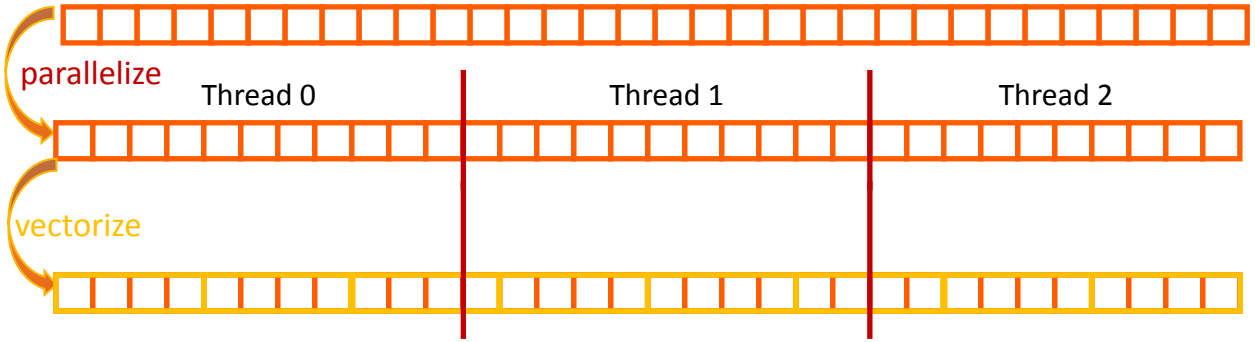
OpenMP SIMD directive

Before(vendor specific)

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

Now(portable)

```
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;
}
```



SIMD function vectorization

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

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

- “declare simd” tells compiler to generate two versions of the function: scalar and vectorized
- Vectorized version will be used in the SIMD regions
- Compiler reports can help to find out if a region is vectorized or not

Questions?
Exercise:

- OpenMP 4.5 version of Jacobi on a BlueWaters
  - copy jacobi-omp45.tar.gz to your home directory
  - tar –xzvf jacobi-omp45.tar.gz
  - cd jacobi-omp45
  - source .modules
  - gmake
  - qsub jacobi.sub
- It uses the following modules:
  - cce/8.4.6
  - craype-accel-nvidia35
  - cudatoolkit
- Example source code
  - jacobi-accel.f

Version 5.0 was released SC'18 (Nov 2018)

- OpenMP 5.0 introduces new powerful features to improve programmability

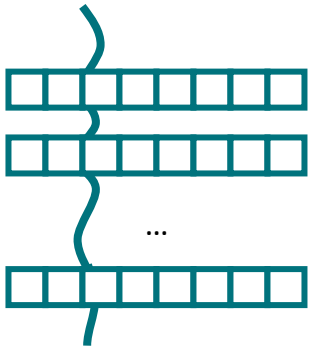
Task Reductions
Memory Allocators
Detachable Tasks
Task-to-data Affinity
Dependence Objects
Tools APIs: OMPD, OMPT
Improved Task Dependencies
Fortran 2008 support
Unified Shared Memory
loop Construct
Display Affinity
C++14 and C++17 support
Multi-level Parallelism
Meta-directives
Collapse non-rect. Loops
Parallel Scan
User Defined Function Variants
Reverse Offloading
Data Serialization for Offload

The New Loop Construct

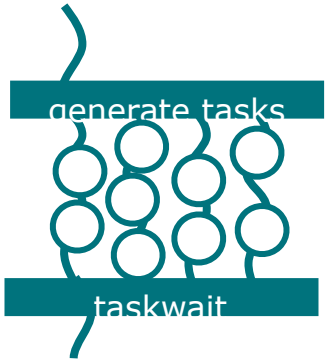
- Existing parallel loop constructs are tightly bound to execution model.

```
#pragma omp for
for (i=0; i<N;++i) {…}
```

#pragma omp simd
for (i=0; i<N;++i) {…}



#pragma omp taskloop
for (i=0; i<N;++i) {…}



The New Loop Construct: Freedom to Compiler

```
#pragma omp loop order(concurrent)
    for (int i = 0; i < n; ++i){
      y[i] = a*x[i] + y[i];
    }
```

- It is meant to give an OpenMP implementation the freedom to choose the best parallelization scheme: teams, threads, taskloop, or SIMD, etc.
- order(concurrent): asserts to the compiler that the iterations of a loop are free of dependencies and may be run concurrently in any order
  - This is the current default
  - Each iteration execute exactly once

Unified Virtual Memory Support

- Single address space over CPU and GPU memories
- Data migrated between CPU and GPU memories transparently to the application - no need to explicitly copy data

```
#pragma omp requires unified_shared_memory
for (k=0; k < NTIMES; k++)
{
// No data directive needed for pointers a, b, c
#pragma omp target teams distribute parallel for
  for (j=0; j<ARRAY_SIZE; j++) {
    a[j] = b[j] + scalar * c[j];
  }
}
```

More Information

- www.openmp.org
- Tutorials:
  - Supercomputing 2019
- User meetings:
  - OpenMPcon 2019
- Workshops
  - IWOMP19, WACCPD'19
- Supercomputing 2019
  - OpenMP Booth
    - Talks, meet the developers, etc.