A “Hands-on” Introduction to OpenMP*

by Tim Mattson Intel Corp.

timothy.g.mattson@intel.com

Introduction

- OpenMP is one of the most common parallel programming models in use today.
- It is relatively easy to use which makes a great language to start with when learning to write parallel software.
- Assumptions:
  - We assume you know C and C++. OpenMP supports Fortran but we will restrict ourselves to C.

Concurrency vs. Parallelism

- Two important definitions:
  - Concurrency: A condition of a system in which multiple tasks are *logically* active at one time.
  - Parallelism: A condition of a system in which multiple tasks are actually active at one time.

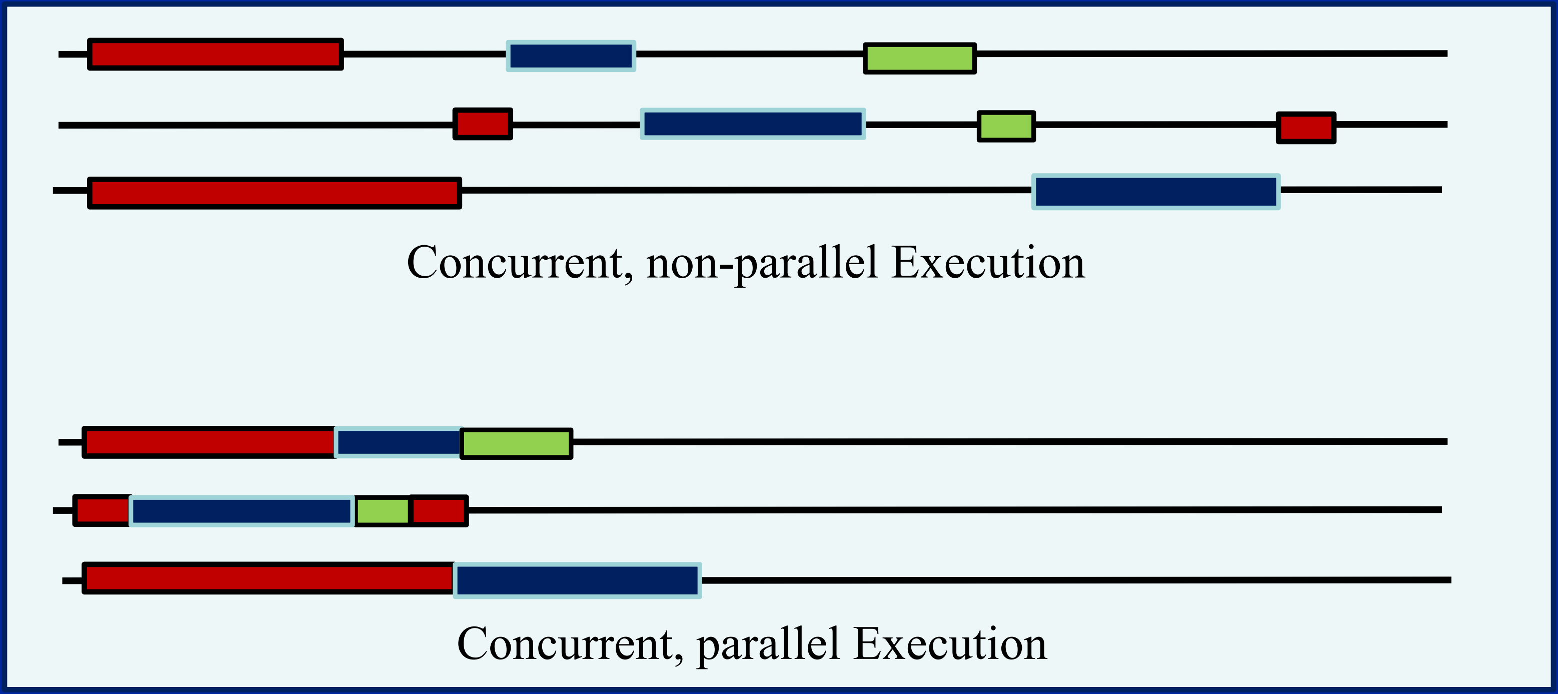

Concurrent, non-parallel Execution
Concurrent, parallel Execution

Figure from "An Introduction to Concurrency in Programming Languages" by J. Sottile, Timothy G. Mattson, and Craig E Rasmussen, 2010

Concurrency vs. Parallelism

- Two important definitions:
  - Concurrency: A condition of a system in which multiple tasks are *logically* active at one time.
  - Parallelism: A condition of a system in which multiple tasks are actually active at one time.

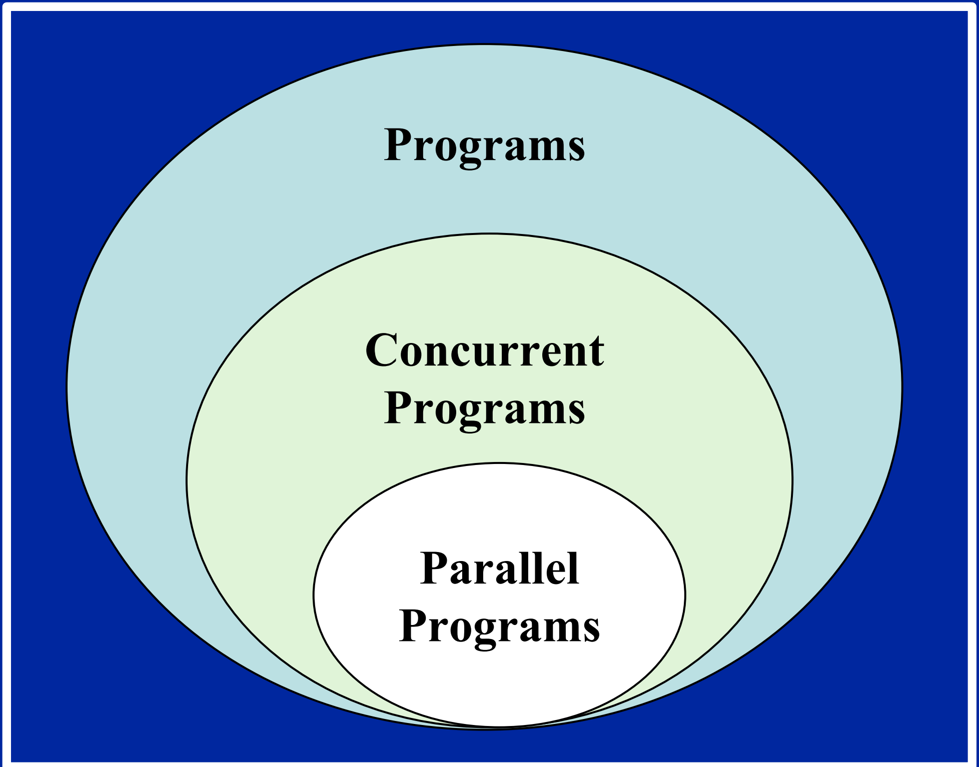


Figure from "An Introduction to Concurrency in Programming Languages" by J. Sottile, Timothy G. Mattson, and Craig E Rasmussen, 2010

Concurrent vs. Parallel applications

- We distinguish between two classes of applications that exploit the concurrency in a problem:
  - Concurrent application: An application for which computations **logically** execute simultaneously due to the semantics of the application.
  - Parallel application: An application for which the computations **actually** execute simultaneously in order to complete a problem in less time.

OpenMP* Overview:

```
C$OMP FLUSH
```

```
#pragma omp critical
```

```
C$OMP THREADPRIVATE(/ABC/)
```

```
CALL OMP_SET_NUM_THREADS(10)
```

```
C$ON
```

```
C$ON
```

```
C$(
```

```
ED
```

```
C
```

```
#p
```

OpenMP: An API for Writing Multithreaded Applications

- A set of compiler directives and library routines for parallel application programmers
- Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++
- Standardizes last 30 years of SMP practice

```
C$OMP PARALLEL COPYIN(/blk/)
```

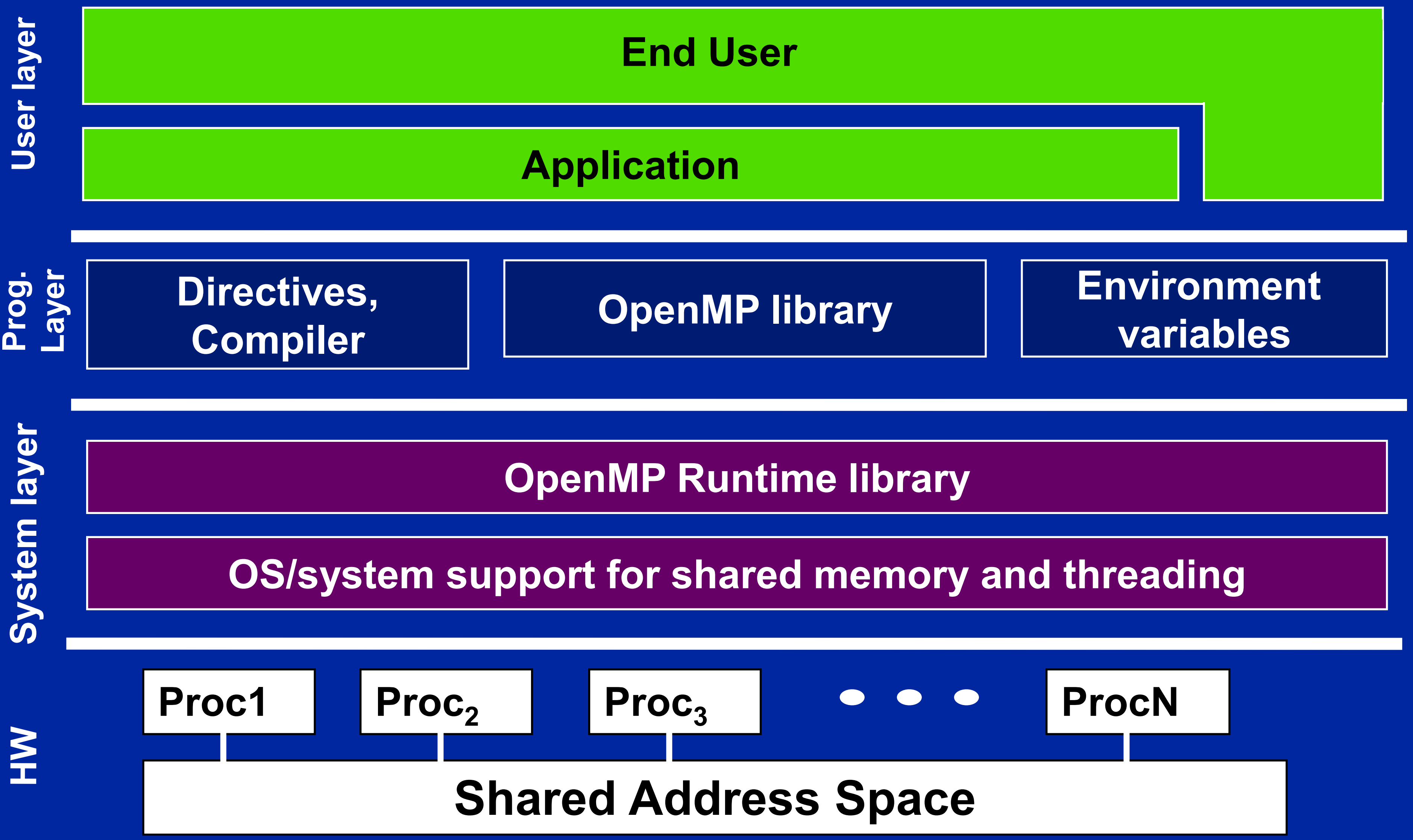
```
C$OMP DO lastprivate(XX)
```

```
Nthrds = OMP_GET_NUM_PROCS()
```

```
omp_set_lock(lck)
```

* The name “OpenMP” is the property of the OpenMP Architecture Review Board.

OpenMP Basic Defs: Solution Stack



User layer

End User

Application

Prog. Layer

Directives, Compiler

OpenMP library

Environment variables

System layer

OpenMP Runtime library

OS/system support for shared memory and threading

HW

Proc1

Proc$_2$

Proc$_3$

• • •

ProcN

Shared Address Space

OpenMP core syntax

- Most of the constructs in OpenMP are compiler directives.

  #pragma omp *construct [clause [clause]…]*

  - Example

    #pragma omp parallel num_threads(4)

- Function prototypes and types in the file:

  #include <omp.h>

- Most OpenMP* constructs apply to a "structured block".
  - Structured block: a block of one or more statements with one point of entry at the top and one point of exit at the bottom.
  - It's OK to have an exit() within the structured block.

Compiler notes: Visual Studio

- Start “new project”
- Select win 32 console project
  - Set name and path
  - On the next panel, Click “next” instead of finish so you can select an empty project on the following panel.
  - Drag and drop your source file into the source folder on the visual studio solution explorer
  - Activate OpenMP
    - Go to project properties/configuration properties/C.C++/language … and activate OpenMP
- Set number of threads inside the program
- Build the project
- Run “without debug” from the debug menu.

Compiler notes: Other

- Linux and OS X with gcc:

for the Bash shell

```
> gcc -fopenmp foo.c
> export OMP_NUM_THREADS=4
> ./a.out
```

Exercise 1, Part A: Hello world

Verify that your environment works

- Write a program that prints “hello world”.

```
int main()
{




    int ID = 0;

    printf(“ hello(%d) ”, ID);
    printf(“ world(%d) \n”, ID);

}
```

Exercise 1, Part B: Hello world

Verify that your OpenMP environment works

- Write a multithreaded program that prints “hello world”.

```
#include <omp.h>
int main()
{
    #pragma omp parallel
    {
      int ID = 0;
      printf(“ hello(%d) ”, ID);
      printf(“ world(%d) \n”, ID);
    }
}
```

Exercise 1: Solution

A multi-threaded “Hello world” program

- Write a multithreaded program where each thread prints “hello world”.

```
#include “omp.h”
int  main()
{

#pragma omp parallel
 {

     int ID = omp_get_thread_num();
     printf(“ hello(%d) ”, ID);
     printf(“ world(%d) \n”, ID);
   }
}
```

OpenMP include file

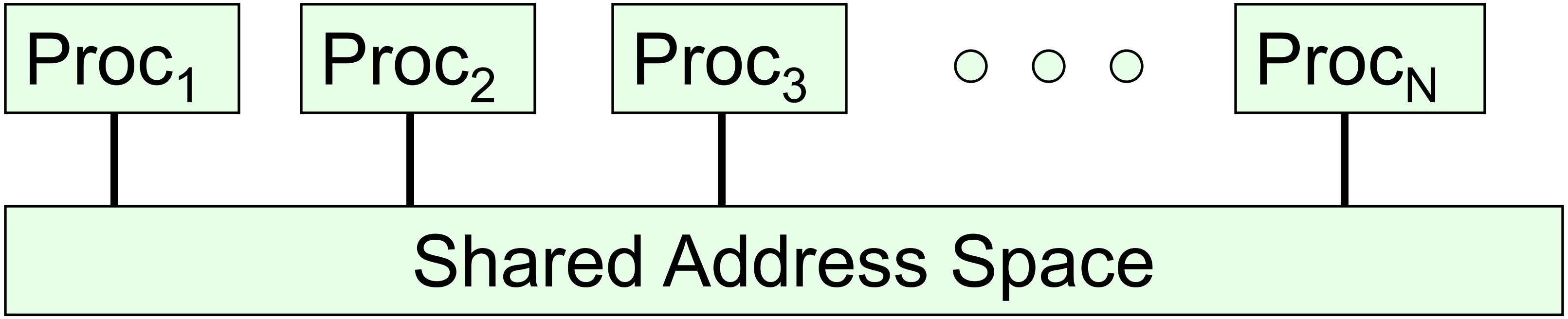
Parallel region with default number of threads

Runtime library function to return a thread ID.

End of the Parallel region

Shared memory Computers

- **Shared memory computer** : any computer composed of multiple processing elements that share an address space. Two Classes:
  - **Symmetric multiprocessor** (**SMP**): a shared address space with "equal-time" access for each processor, and the OS treats every processor the same way.
  - **Non Uniform address space multiprocessor** (**NUMA**): different memory regions have different access costs … think of memory segmented into "Near" and "Far" memory.

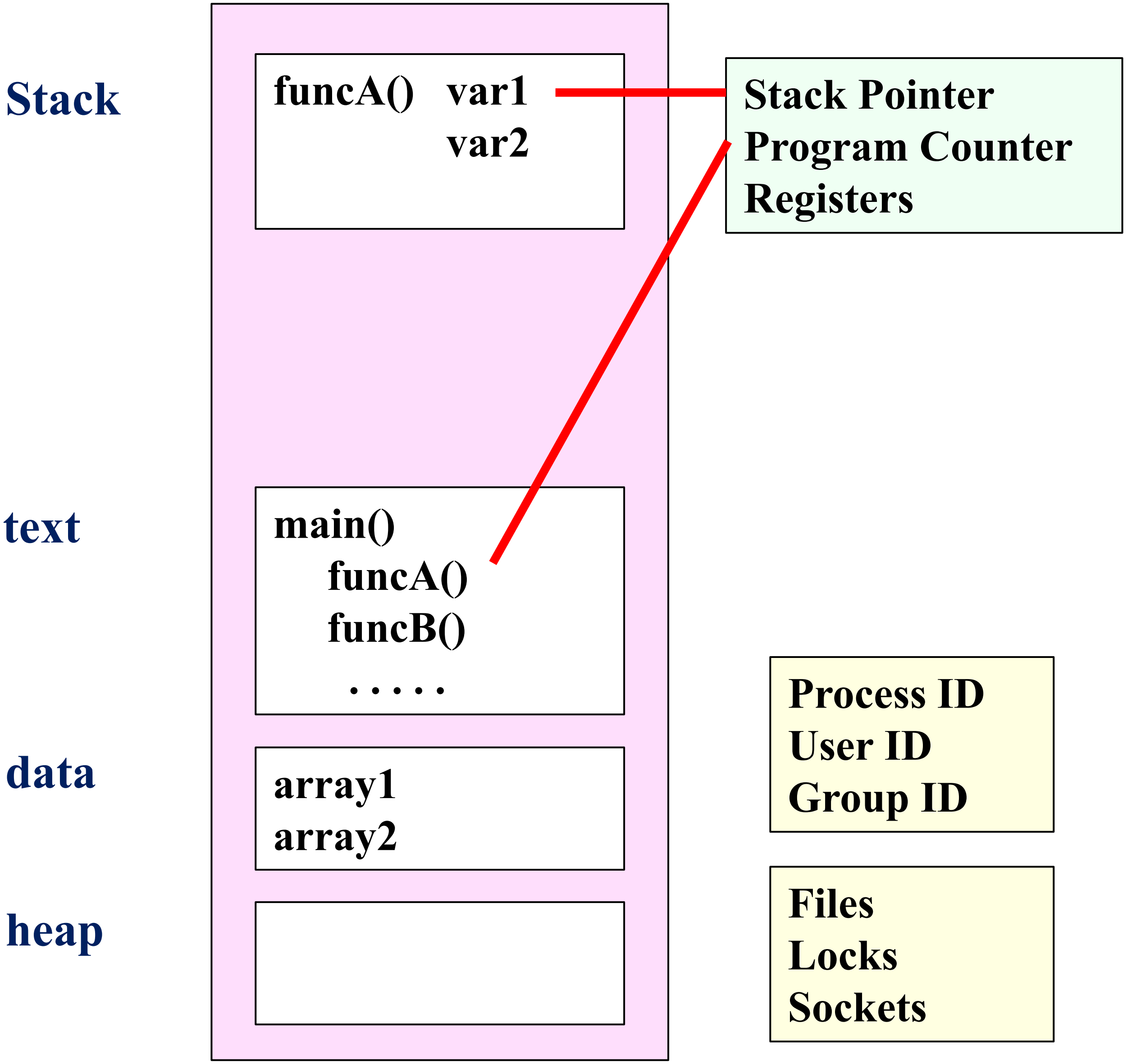


Shared memory computers

- The shared address space and (as we will see) programming models encourage us to think of them at SMP systems.

- Reality is more complex … any multiprocessor CPU with a cache is a NUMA system. Start out by treating the system as an SMP and just accept that much of your optimization work will address cases where that case breaks down.

Programming shared memory computers



Stack

funcA() var1
var2

Stack Pointer
Program Counter
Registers

text

main()
funcA()
funcB()
.....

data

array1
array2

heap

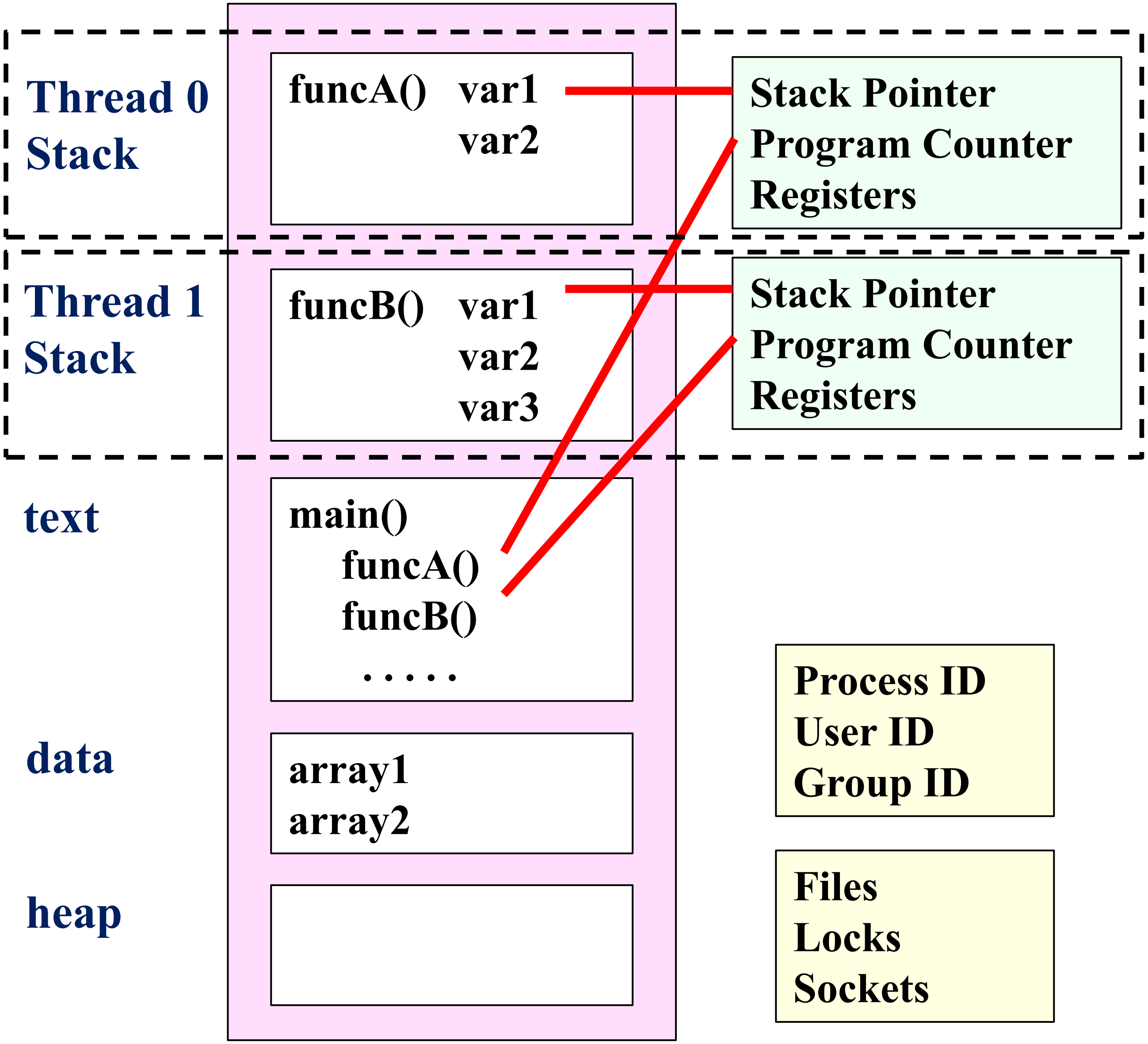
Process ID
User ID
Group ID

Files
Locks
Sockets

Process

- An instance of a program execution.
- The execution context of a running program … i.e. the resources associated with a program’s execution.

Programming shared memory computers



Thread 0 Stack

funcA() var1
var2

Stack Pointer
Program Counter
Registers

Thread 1 Stack

funcB() var1
var2
var3

Stack Pointer
Program Counter
Registers

text

main()
funcA()
funcB()
.....

data

array1
array2

heap

Process ID
User ID
Group ID

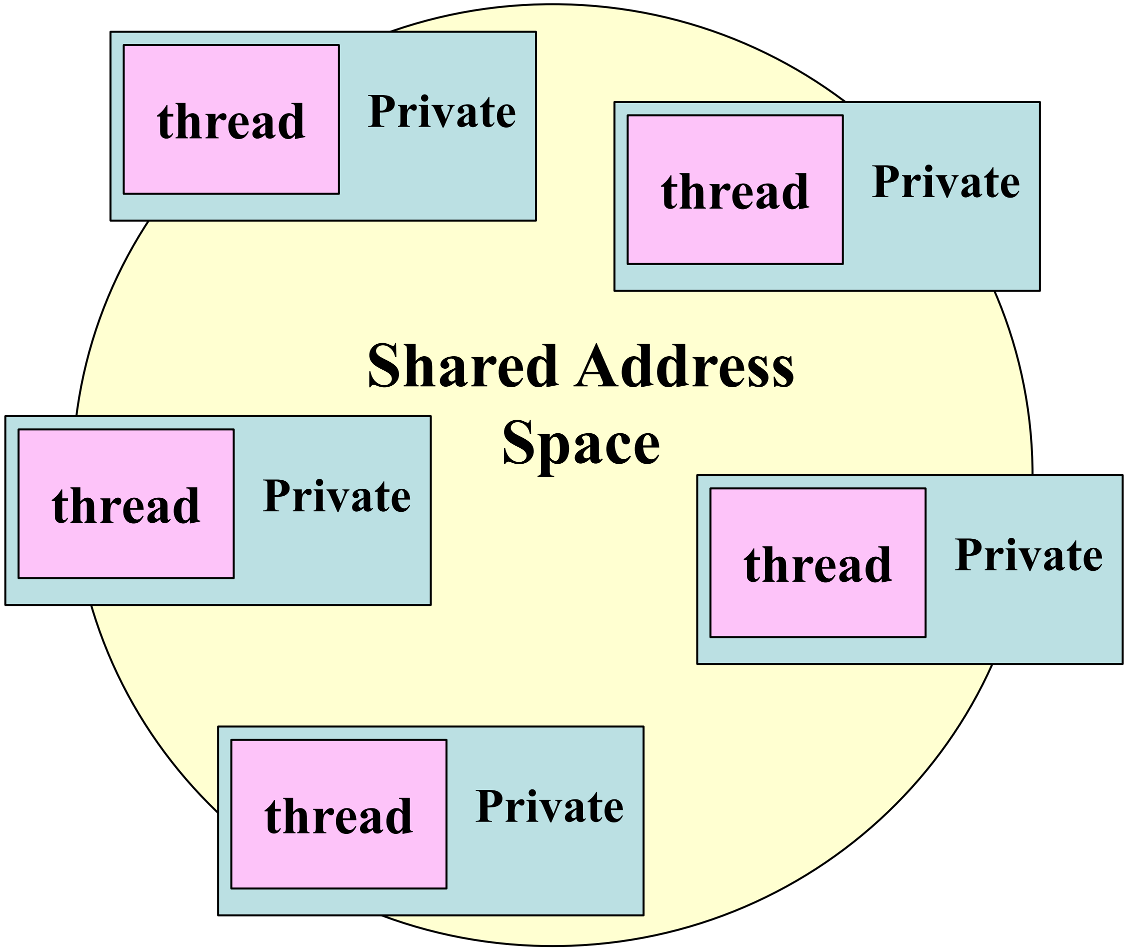
Files
Locks
Sockets

Threads:

- Threads are "light weight processes”
- Threads share Process state among multiple threads … this greatly reduces the cost of switching context.

A shared memory program

- **An instance of a program:**
  - One process and lots of threads.
  - Threads interact through reads/writes to a shared address space.
  - OS scheduler decides when to run which threads … interleaved for fairness.
  - Synchronization to assure every legal order results in correct results.



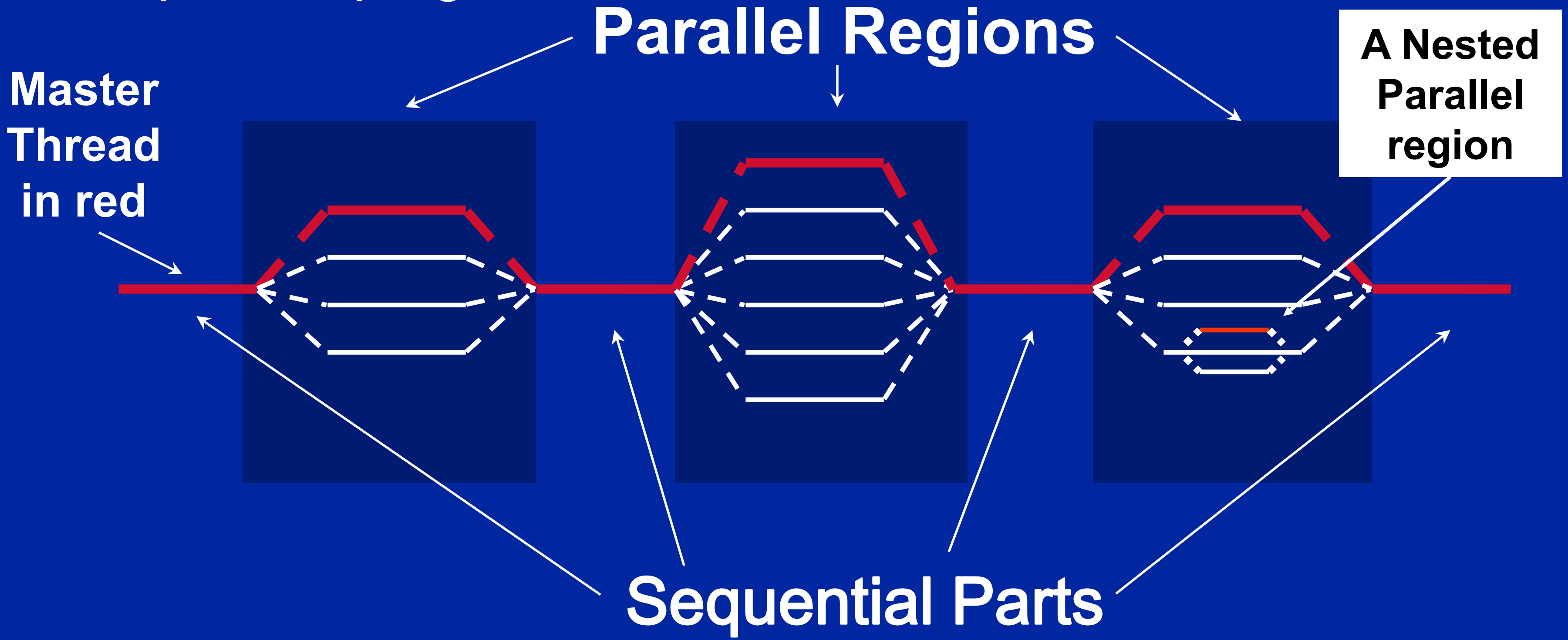
OpenMP Overview:
How do threads interact?

- OpenMP is a multi-threading, shared address model:
  - Threads communicate by sharing variables.
- Unintended sharing of data causes race conditions:
  - *race condition*: when the program’s outcome changes as the threads are scheduled differently.
- To control race conditions:
  - Use synchronization to protect data conflicts.
- Synchronization is expensive so:
  - Change how data is accessed to minimize the need for synchronization.

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.



Thread Creation: Parallel Regions

- **You create threads in OpenMP* with the parallel construct.**
- **For example, To create a 4 thread Parallel region:**

Each thread executes a copy of the code within the structured block

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
        int id = omp_get_thread_num();
        pooh(id, A);
}
```

Runtime function to request a certain number of threads

Runtime function returning a thread ID

- **Each thread calls pooh(id, A) for `id` = 0 to 3**

* The name "OpenMP" is the property of the OpenMP Architecture Review Board

Thread Creation: Parallel Regions

- You create threads in OpenMP* with the parallel construct.
- For example, To create a 4 thread Parallel region:

Each thread executes a copy of the code within the structured block

```
double A[1000];

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

clause to request a certain number of threads

Runtime function returning a thread ID

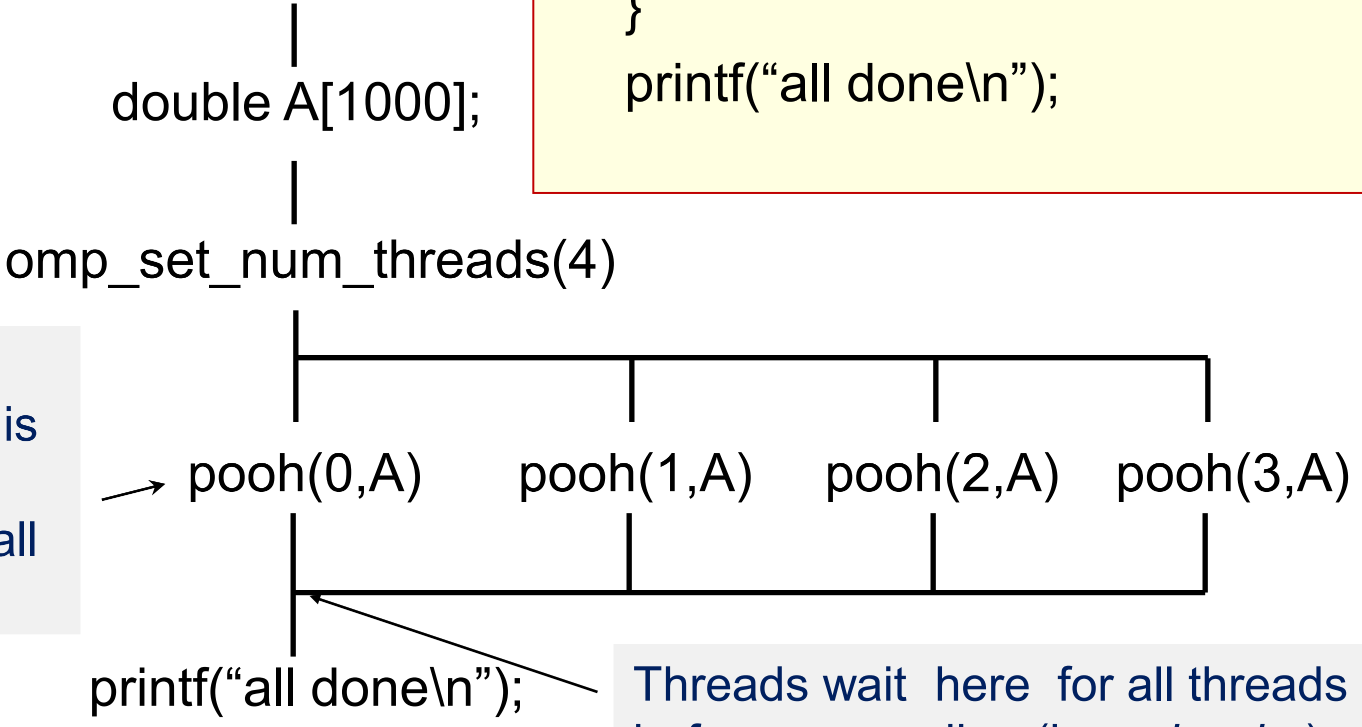
- Each thread calls pooh(id, A) for `id` = `0` to `3`

* The name “OpenMP” is the property of the OpenMP Architecture Review Board

Thread Creation: Parallel Regions

- Each thread executes the same code redundantly.

```
double A[1000];
#pragma omp parallel num_threads(4)
{
    int id = omp_get_thread_num();
    pooh(id, A);
}
printf(“all done\n”);
```



double A[1000];

omp_set_num_threads(4)

pooh(0,A) pooh(1,A) pooh(2,A) pooh(3,A)

printf(“all done\n”);

A single copy of A is shared between all threads.

Threads wait here for all threads to finish before proceeding (i.e. a *barrier*)

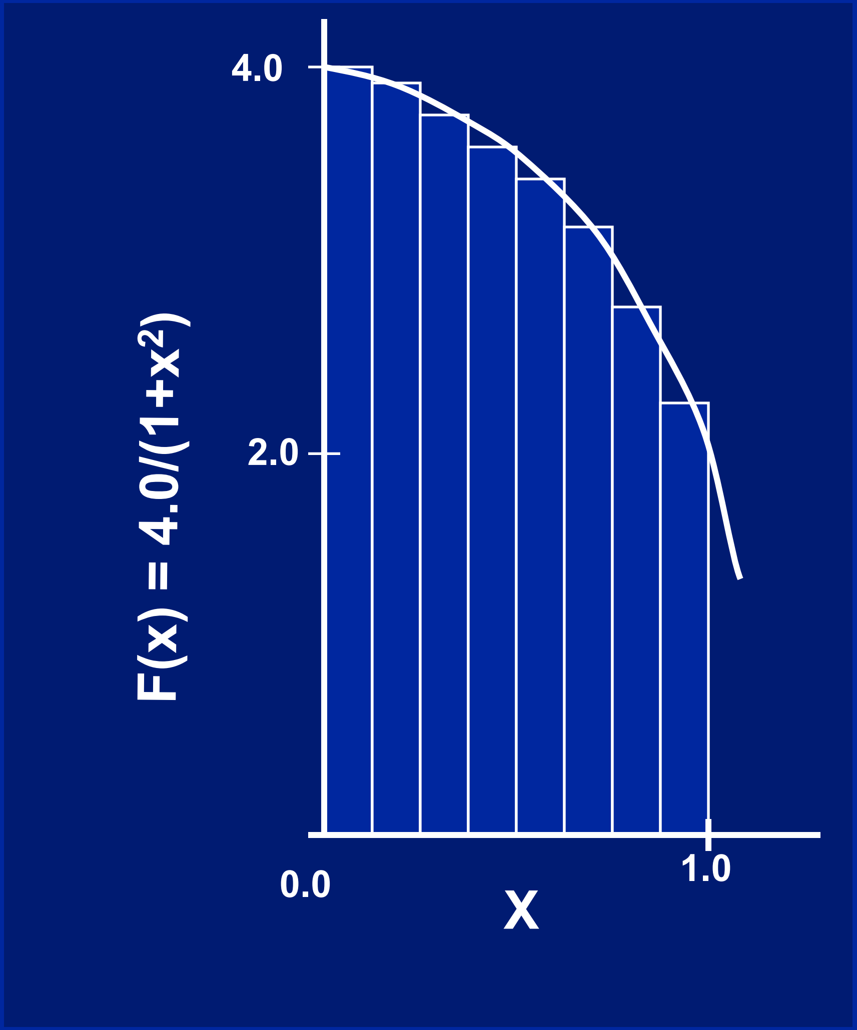
OpenMP: what the compiler does

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

- The OpenMP compiler generates code logically analogous to that on the right of this slide, given an OpenMP pragma such as that on the top-left
- All known OpenMP implementations use a thread pool so full cost of threads creation and destruction is not incurred for reach parallel region.
- Only three threads are created because the last parallel section will be invoked from the parent thread.

```
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]);
```

Exercise:
Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\, dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Exercise: Serial PI Program

```
static long num_steps = 100000;
double step;
int main ()
{           int i;    double x, pi, sum = 0.0;

            step = 1.0/(double) num_steps;

            for (i=0;i< num_steps; i++){
                      x = (i+0.5)*step;
                      sum = sum + 4.0/(1.0+x*x);
            }
            pi = step * sum;
}
```

Exercise:

- Create a parallel version of the pi program using a parallel construct.
- Pay close attention to shared versus private variables.
- In addition to a parallel construct, you will need the runtime library routines
  - int omp_get_num_threads();
  - int omp_get_thread_num();
  - double omp_get_wtime();

Number of threads in the team

Thread ID or rank

Time in Seconds since a fixed point in the past

Serial PI Program

```
static long num_steps = 100000;
double step;
int main ()
{          int i;   double x, pi, sum = 0.0;

           step = 1.0/(double) num_steps;

           for (i=0;i< num_steps; i++){
                  x = (i+0.5)*step;
                  sum = sum + 4.0/(1.0+x*x);
           }
           pi = step * sum;
}
```

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;
}
```

Promote scalar to an array dimensioned by number of threads to avoid race condition.

Only one thread should copy the number of threads to the global value to make sure multiple threads writing to the same address don’t conflict.

This is a common trick in SPMD programs to create a cyclic distribution of loop iterations

Algorithm strategy:

The SPMD (Single Program Multiple Data) design pattern

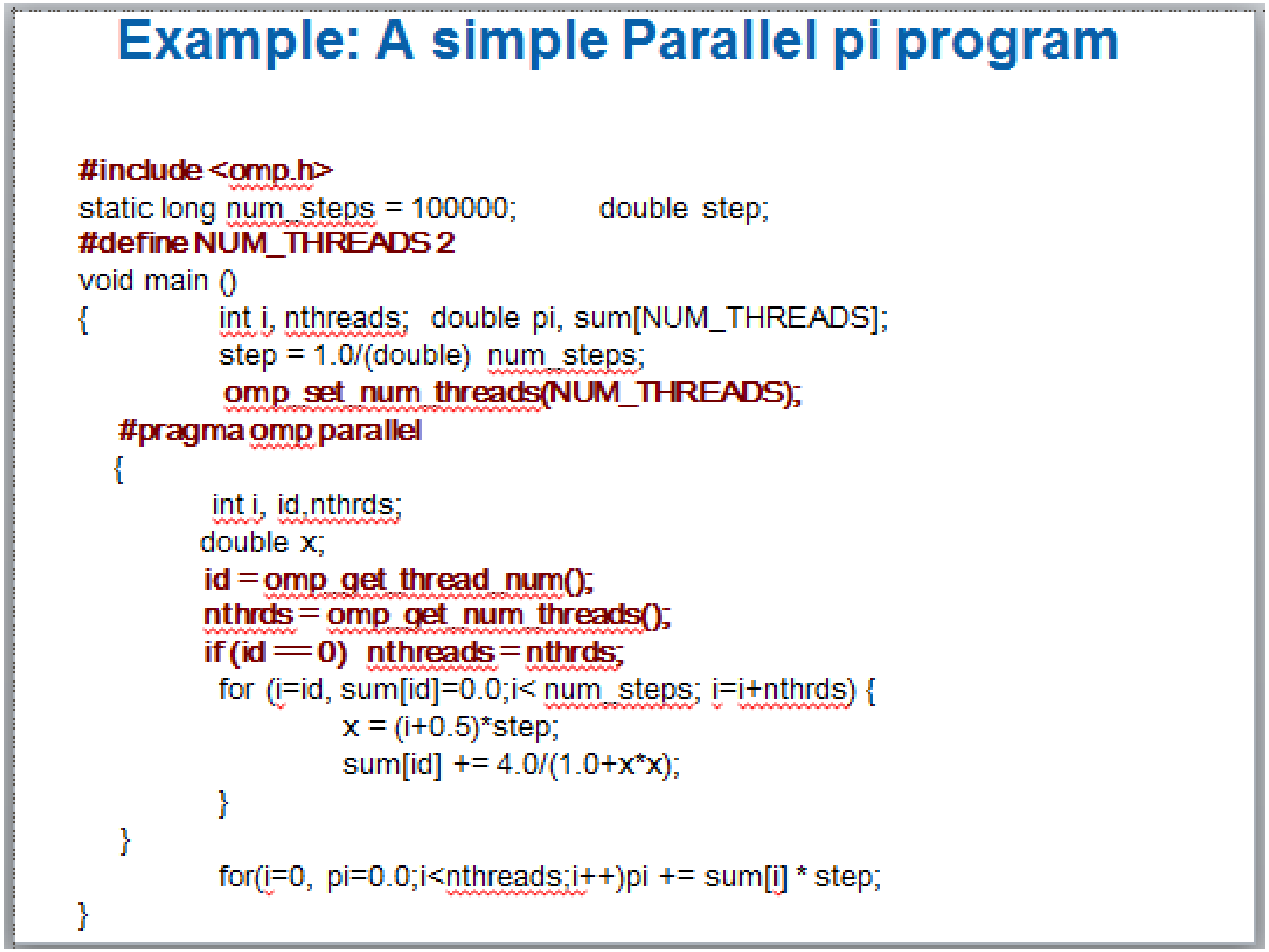
- Run the same program on P processing elements where P can be arbitrarily large.
- Use the rank … an ID ranging from 0 to (P-1) … to select between a set of tasks and to manage any shared data structures.

This pattern is very general and has been used to support most (if not all) the algorithm strategy patterns.

MPI programs almost always use this pattern … it is probably the most commonly used pattern in the history of parallel programming.

Results*

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

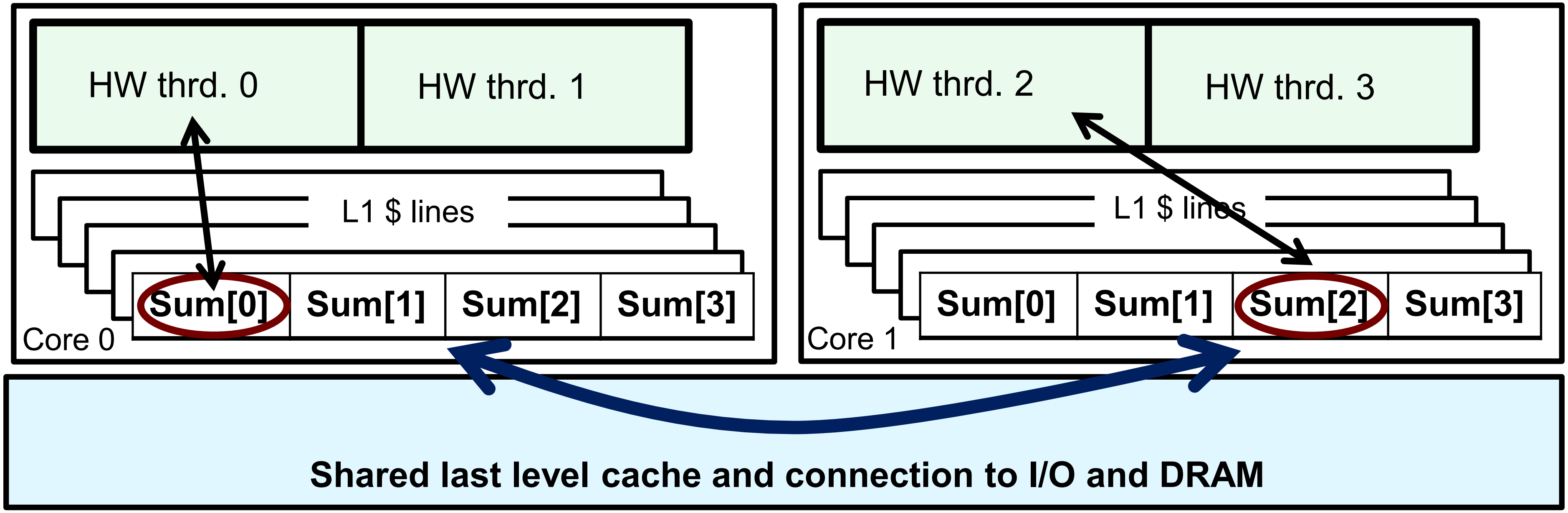


Example: A simple Parallel pi program

```
#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;
}
```

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.

Example: eliminate False sharing by padding the sum array

```
#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;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

Do we really need to pad our arrays?

- Padding arrays requires deep knowledge of the cache architecture (not true). Move to a machine with different sized cache lines and your software performance falls apart.
- There has got to be a better way to deal with false sharing.

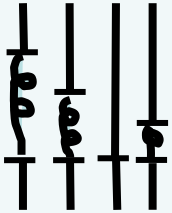
OpenMP Overview:
How do threads interact?

Recall our high level overview of OpenMP?

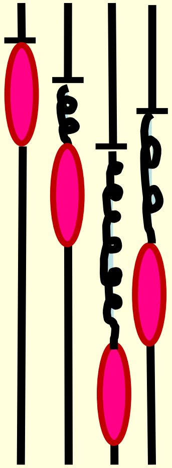
- OpenMP is a multi-threading, shared address model.
  – Threads communicate by sharing variables.
- Unintended sharing of data causes race conditions:
  – race condition: when the program's outcome changes as the threads are scheduled differently.
- To control race conditions:
  – Use synchronization to protect data conflicts.
- Synchronization is expensive so:
  – Change how data is accessed to minimize the need for synchronization.

Synchronization:

- Synchronization: bringing one or more threads to a well defined and known point in their execution.
- The two most common forms of synchronization are:



Barrier: each thread wait at the barrier until all threads arrive.



Mutual exclusion: Define a block of code that only one thread at a time can execute.

Synchronization

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

- High level synchronization:
  - critical
  - atomic
  - barrier
  - ordered
- Low level synchronization
  - flush
  - locks (both simple and nested)

Discussed later

Synchronization: Barrier

- **Barrier**: Each thread waits until all threads arrive.

```
#pragma omp parallel

{
          int id=omp_get_thread_num();
          A[id] = big_calc1(id);
#pragma omp barrier


          B[id] = big_calc2(id, A);
}
```

Synchronization: critical

- Mutual exclusion: Only one thread at a time can enter a **critical** region.

```
float  res;
#pragma omp parallel
{     float B;   int i, id, nthrds;
      id = omp_get_thread_num();
      nthrds = omp_get_num_threads();
       for(i=id;i<niters;i+=nthrds){
             B =  big_job(i);
#pragma omp critical
             res += consume (B);
      }
}
```

Threads wait their turn – only one at a time calls consume()

Synchronization: Atomic (basic form)

- **Atomic** provides mutual exclusion but only applies to the update of a memory location (the update of X in the following example)

```
#pragma omp parallel

{
         double tmp, B;

        B =  DOIT();

        tmp = big_ugly(B);

 #pragma omp atomic
           X +=  tmp;

}
```

The statement inside the atomic must be one of the following forms:

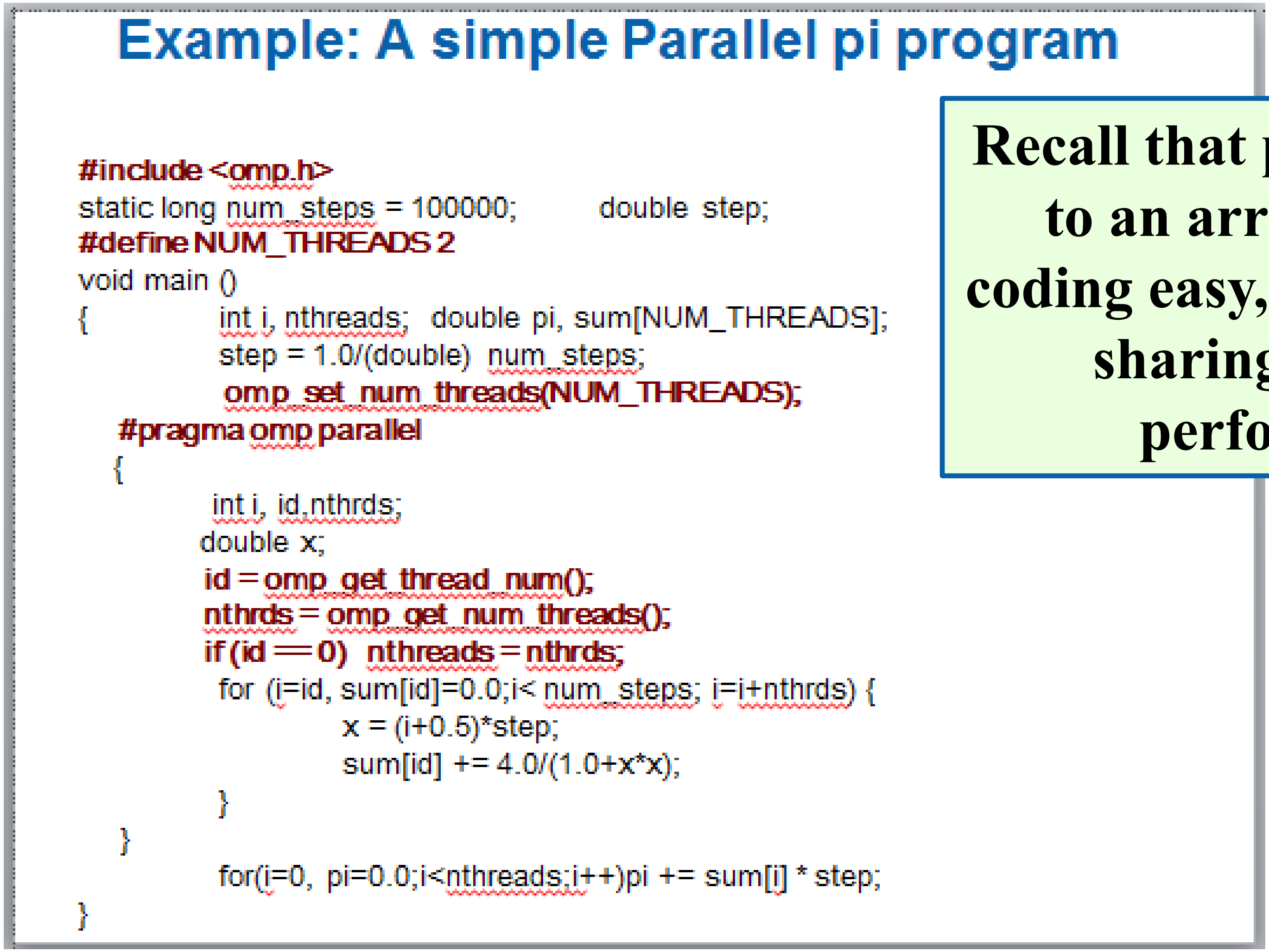
- x binop= expr
- x++
- ++x
- x—
- --x

X is an lvalue of scalar type and binop is a non-overloaded built in operator.

Exercise (cont)

- In exercise 2, you used an array to create space for each thread to store its partial sum.
- If array elements happen to share a cache line, this leads to false sharing.
  - Non-shared data in the same cache line so each update invalidates the cache line … in essence “sloshing independent data” back and forth between threads.
- Modify your “pi program” from exercise 2 to avoid false sharing due to the sum array.

Pi program with false sharing*



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;
}
```

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

Example: Using a critical section to remove impact of false sharing

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

Create a scalar local to each thread to accumulate partial sums.

No array, so no false sharing.

Sum goes "out of scope" beyond the parallel region … so you must sum it in here. Must protect summation into pi in a critical region so updates don't conflict

Example: Using a critical section to remove impact of false sharing

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

Be careful where you put a critical section

What would happen if you put the critical section inside the loop?

Example: Using an atomic to remove impact of false sharing

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

Create a scalar local to each thread to accumulate partial sums.

No array, so no false sharing.

Sum goes “out of scope” beyond the parallel region … so you must sum it in here. Must protect summation into pi so updates don’t conflict

SPMD vs. worksharing

- A parallel construct by itself creates an SPMD or "Single Program Multiple Data" program … i.e., each thread redundantly executes the same code.
- How do you split up pathways through the code between threads within a team?
  - This is called worksharing
    - Loop construct
    - Sections/section constructs
    - Single construct
    - Task construct

Discussed later

The loop worksharing Constructs

- The loop worksharing construct splits up loop iterations among the threads in a team

```
#pragma omp parallel
{
#pragma omp for
        for (I=0;I<N;I++){
                NEAT_STUFF(I);
        }
}
```

Loop construct name:

- C/C++: for
- Fortran: do

The variable I is made "private" to each thread by default. You could do this explicitly with a "private(I)" clause

Loop worksharing Constructs

A motivating example

Sequential code

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

OpenMP parallel region

```
#pragma omp parallel
{
            int id, i, Nthrds, istart, iend;
            id = omp_get_thread_num();
            Nthrds = omp_get_num_threads();
            istart = id * N / Nthrds;
            iend = (id+1) * N / Nthrds;
            if (id == Nthrds-1)iend = N;
            for(i=istart;i<iend;i++)   { a[i] = a[i] + b[i];}
}
```

OpenMP parallel region and a worksharing for construct

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

loop worksharing constructs: The schedule clause

- **The schedule clause affects how loop iterations are mapped onto threads**
  - schedule(static [,chunk])
    - Deal-out blocks of iterations of size “chunk” to each thread.
  - schedule(dynamic[,chunk])
    - Each thread grabs “chunk” iterations off a queue until all iterations have been handled.
  - schedule(guided[,chunk])
    - Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size “chunk” as the calculation proceeds.
  - schedule(runtime)
    - Schedule and chunk size taken from the OMP_SCHEDULE environment variable (or the runtime library).
  - schedule(auto)
    - Schedule is left up to the runtime to choose (does not have to be any of the above).

loop work-sharing constructs:
The schedule clause

Schedule Clause	When To Use
STATIC	Pre-determined and predictable by the programmer
DYNAMIC	Unpredictable, highly variable work per iteration
GUIDED	Special case of dynamic to reduce scheduling overhead
AUTO	When the runtime can "learn" from previous executions of the same loop

Least work at runtime : scheduling done at compile-time

Most work at runtime : complex scheduling logic used at run-time

Combined parallel/worksharing construct

- OpenMP shortcut: Put the “parallel” and the worksharing directive on the same line

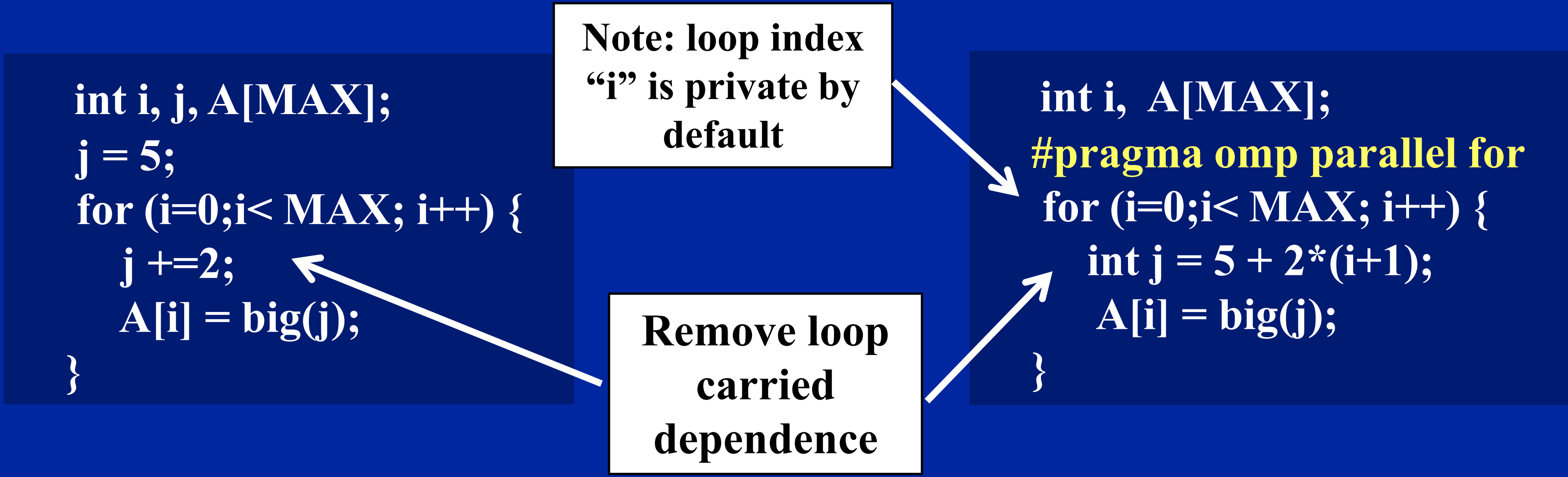
```
 double  res[MAX];  int i;
#pragma omp parallel
{
   #pragma omp for
   for (i=0;i< MAX; i++) {
       res[i] = huge();
   }
}
```

```
 double  res[MAX];  int i;
#pragma omp parallel for
   for (i=0;i< MAX; i++) {
       res[i] = huge();
   }
```

These are equivalent

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

Nested loops

- For perfectly nested rectangular loops we can parallelize multiple loops in the nest with the collapse clause:

```
#pragma omp parallel for collapse(2)
for (int i=0; i<N; i++) {
   for (int j=0; j<M; j++) {
            .....
   }
}
```

Number of loops to be parallelized, counting from the outside

- Will form a single loop of length NxM and then parallelize that.
- Useful if N is O(no. of threads) so parallelizing the outer loop makes balancing the load difficult.

Reduction

- How do we handle this case?

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

- We are combining values into a single accumulation variable (ave) … there is a true dependency between loop iterations that can’t be trivially removed
- This is a very common situation … it is called a “reduction”.
- Support for reduction operations is included in most parallel programming environments.

Reduction

- OpenMP reduction clause:

  reduction (op : list)

- Inside a parallel or a work-sharing construct:
  - A local copy of each list variable is made and initialized depending on the “op” (e.g. 0 for “+”).
  - Updates occur on the local copy.
  - Local copies are reduced into a single value and combined with the original global value.
- The variables in “list” must be shared in the enclosing parallel region.

```
 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;
```

OpenMP: Reduction operands/initial-values

- Many different associative operands can be used with reduction:
- Initial values are the ones that make sense mathematically.

Operator	Initial value
+	0
*	1
-	0
min	Largest pos. number
max	Most neg. number

C/C++ only	
Operator	Initial value
&	~0
|	0
^	0
&&	1
||	0

Fortran Only	
Operator	Initial value
.AND.	.true.
.OR.	.false.
.NEQV.	.false.
.IEOR.	0
.IOR.	0
.IAND.	All bits on
.EQV.	.true.

Exercise (cont): Pi with loops

- Go back to the serial pi program and parallelize it with a loop construct
- Your goal is to minimize the number of changes made to the serial program.

Example: Pi with a loop and a reduction

```
#include <omp.h>
static long num_steps = 100000;         double step;
void main ()
{    int i;          double x, pi, sum = 0.0;
     step = 1.0/(double) num_steps;
     #pragma omp parallel
     {
         double x;
       #pragma omp for 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;
}
```

Create a team of threads … without a parallel construct, you'll never have more than one thread

Create a scalar local to each thread to hold value of x at the center of each interval

Break up loop iterations and assign them to threads … setting up a reduction into sum. Note … the loop indix is local to a thread by default.

Synchronization: Barrier

- **Barrier**: Each thread waits until all threads arrive.

```
#pragma omp parallel shared (A, B, C) private(id)
{
        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

no implicit barrier due to nowait

implicit barrier at the end of a parallel region

Master Construct

- The **master** construct denotes a structured block that is only executed by the master thread.
- The other threads just skip it (no synchronization is implied).

```
#pragma omp parallel
{
        do_many_things();
#pragma omp master
        {    exchange_boundaries();   }
#pragma omp  barrier
        do_many_other_things();
}
```

Single worksharing Construct

- The single construct denotes a block of code that is executed by only one thread (not necessarily the master thread).
- A barrier is implied at the end of the single block (can remove the barrier with a *nowait* clause).

```
#pragma omp parallel
{
        do_many_things();
#pragma omp single
        {    exchange_boundaries();   }
        do_many_other_things();
}
```

Sections worksharing Construct

- The *Sections* worksharing construct gives a different structured block to each thread.

```
#pragma omp parallel
{

   #pragma omp sections
   {
   #pragma omp section
            X_calculation();
   #pragma omp section
             y_calculation();
   #pragma omp section
             z_calculation();
   }

}
```

By default, there is a barrier at the end of the “omp sections”. Use the “nowait” clause to turn off the barrier.

Synchronization: Lock routines

A lock implies a memory fence (a “flush”) of all thread visible variables

- Simple Lock routines:
  - A simple lock is available if it is unset.
    - omp_init_lock(), omp_set_lock(), omp_unset_lock(), omp_test_lock(), omp_destroy_lock()
- Nested Locks
  - A nested lock is available if it is unset or if it is set but owned by the thread executing the nested lock function
    - omp_init_nest_lock(), omp_set_nest_lock(), omp_unset_nest_lock(), omp_test_nest_lock(), omp_destroy_nest_lock()

Note: a thread always accesses the most recent copy of the lock, so you don’t need to use a flush on the lock variable.

Synchronization: Simple Locks

- Example: conflicts are rare, but to play it safe, we must assure mutual exclusion for updates to histogram elements.

```
#pragma omp parallel for
 for(i=0;i<NBUCKETS; i++){
      omp_init_lock(&hist_locks[i]);    hist[i] = 0;
 }
 #pragma omp parallel for
 for(i=0;i<NVALS;i++){
    ival = (int)  sample(arr[i]);
    omp_set_lock(&hist_locks[ival]);
         hist[ival]++;
    omp_unset_lock(&hist_locks[ival]);
  }

for(i=0;i<NBUCKETS; i++)
  omp_destroy_lock(&hist_locks[i]);
```

One lock per element of hist

Enforce mutual exclusion on update to hist array

Free-up storage when done.

Runtime Library routines

- Runtime environment routines:
  - Modify/Check the number of threads
    - omp_set_num_threads(), omp_get_num_threads(), omp_get_thread_num(), omp_get_max_threads()
  - Are we in an active parallel region?
    - omp_in_parallel()
  - Do you want the system to dynamically vary the number of threads from one parallel construct to another?
    - omp_set_dynamic, omp_get_dynamic();
  - How many processors in the system?
    - omp_num_procs()

…plus a few less commonly used routines.

Runtime Library routines

- **To use a known, fixed number of threads in a program, (1) tell the system that you don't want dynamic adjustment of the number of threads, (2) set the number of threads, then (3) save the number you got.**

```
#include <omp.h>
void main()
{   int num_threads;
      omp_set_dynamic( 0 );
      omp_set_num_threads( omp_num_procs() );
#pragma omp parallel
   {      int id=omp_get_thread_num();
#pragma omp single
              num_threads = omp_get_num_threads();
           do_lots_of_stuff(id);
    }
}
```

Disable dynamic adjustment of the number of threads.

Request as many threads as you have processors.

Protect this op since Memory stores are not atomic

Even in this case, the system may give you fewer threads than requested. If the precise # of threads matters, test for it and respond accordingly.

Environment Variables

- Set the default number of threads to use.
  - **OMP_NUM_THREADS *int_literal***
- OpenMP added an environment variable to control the size of child threads' stack
  - **OMP_STACKSIZE**
- Also added an environment variable to hint to runtime how to treat idle threads
  - **OMP_WAIT_POLICY**
    - **ACTIVE keep threads alive at barriers/locks**
    - **PASSIVE try to release processor at barriers/locks**
- Process binding is enabled if this variable is true … i.e. if true the runtime will not move threads around between processors.
  - **OMP_PROC_BIND true | false**

Data environment:
Default storage attributes

- Shared Memory programming model:
  - Most variables are shared by default
- Global variables are SHARED among threads
  - Fortran: COMMON blocks, SAVE variables, MODULE variables
  - C: File scope variables, static
  - Both: dynamically allocated memory (ALLOCATE, malloc, new)
- But not everything is shared...
  - Stack variables in subprograms(Fortran) or functions(C) called from parallel regions are PRIVATE
  - Automatic variables within a statement block are PRIVATE.

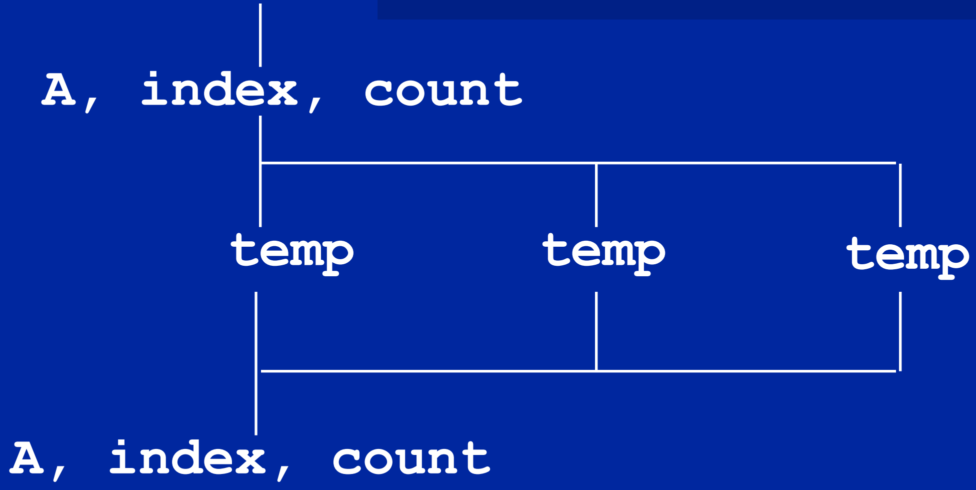
Data sharing: Examples

```
 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



Data sharing:
Changing storage attributes

- One can selectively change storage attributes for constructs using the following clauses*
  - SHARED
  - PRIVATE
  - FIRSTPRIVATE

All the clauses on this page apply to the OpenMP construct NOT to the entire region.

- The final value of a private inside a parallel loop can be transmitted to the shared variable outside the loop with:
  - LASTPRIVATE
- The default attributes can be overridden with:
  - DEFAULT (PRIVATE | SHARED | NONE)

    DEFAULT(PRIVATE) *is Fortran only*

*All data clauses apply to parallel constructs and worksharing constructs except "shared" which only applies to parallel constructs.

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 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

Firstprivate Clause

- Variables initialized from shared variable
- C++ objects are copy-constructed

```
incr = 0;
#pragma omp parallel for firstprivate(incr)
for (i = 0; i <= MAX; i++) {
        if ((i%2)==0) incr++;
        A[i] = incr;
}
```

Each thread gets its own copy of incr with an initial value of 0

Lastprivate Clause

- Variables update shared variable using value from last iteration
- C++ objects are updated as if by assignment

```
void sq2(int n, double *lastterm)
{
    double x; int i;
    #pragma omp parallel for lastprivate(x)
    for (i = 0; i < n; i++){
        x = a[i]*a[i] + b[i]*b[i];
        b[i] = sqrt(x);
     }
     *lastterm = x;
}
```

"x" has the value it held for the "last sequential" iteration (i.e., for i=(n-1))

Data Sharing:

A data environment test

- **Consider this example of PRIVATE and FIRSTPRIVATE**

```
            variables:  A = 1,B = 1, C = 1
#pragma omp parallel private(B)  firstprivate(C)
```

- **Are A,B,C local to each thread or shared inside the parallel region?**
- **What are their initial values inside and values after the parallel region?**

Inside this parallel region ...

- **“A” is shared by all threads; equals 1**
- **“B” and “C” are local to each thread.**
  - **B’s initial value is undefined**
  - **C’s initial value equals 1**

Following the parallel region ...

- **B and C revert to their original values of 1**
- **A is either 1 or the value it was set to inside the parallel region**

Data Sharing: Default Clause

- Note that the default storage attribute is **DEFAULT(SHARED)** (so no need to use it)
  - Exception: #pragma omp task
- To change default: **DEFAULT(PRIVATE)**
  - *each* variable in the construct is made private as if specified in a private clause
  - mostly saves typing
- **DEFAULT(NONE)**: *no* default for variables in static extent. Must list storage attribute for each variable in static extent. Good programming practice!

Only the Fortran API supports default(private).

C/C++ only has default(shared) or default(none).

Serial PI Program

Now that you understand how to modify the data environment, let's take one last look at our pi program.

```
static long num_steps = 100000;
double step;
int main ()
{           int i;   double x, pi, sum = 0.0;

            step = 1.0/(double) num_steps;

            for (i=0;i< num_steps; i++){
                        x = (i+0.5)*step;
                        sum = sum + 4.0/(1.0+x*x);
            }
            pi = step * sum;
}
```

What is the minimum change I can make to this code to parallelize it?

Example: Pi program … minimal changes

```
#include <omp.h>
static long num_steps = 100000;        double step;

void 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;
}
```

For good OpenMP implementations, reduction is more scalable than critical.

i private by default

Note: we created a parallel program without changing any executable code and by adding 2 simple lines of text!

Major OpenMP constructs we've covered so far

- To create a team of threads
  - #pragma omp parallel
- To share work between threads:
  - #pragma omp for
  - #pragma omp single
- To prevent conflicts (prevent races)
  - #pragma omp critical
  - #pragma omp atomic
  - #pragma omp barrier
  - #pragma omp master
- Data environment clauses
  - private (variable_list)
  - firstprivate (variable_list)
  - lastprivate (variable_list)
  - reduction(+:variable_list)

Where variable_list is a comma separated list of variables

Print the value of the macro

_OPENMP

And its value will be

yyyymm

For the year and month of the spec the implementation used

Consider simple list traversal

- Given what we’ve covered about OpenMP, how would you process this loop in Parallel?

```
p=head;
while (p) {
    process(p);
    p = p->next;
}
```

- Remember, the loop worksharing construct only works with loops for which the number of loop iterations can be represented by a closed-form expression at compiler time. While loops are not covered.

list traversal

- When we first created OpenMP, we focused on common use cases in HPC … Fortran arrays processed over “regular” loops.
- Recursion and “pointer chasing” were so far removed from our Fortan focus that we didn’t even consider more general structures.
- Hence, even a simple list traversal is exceedingly difficult with the original versions of OpenMP.

```
p=head;
while (p) {
   process(p);
   p = p->next;
}
```

Linked lists without tasks

```
while (p != NULL) {
    p = p->next;
     count++;
}
p = head;
for(i=0; i<count; i++) {
     parr[i] = p;
     p = p->next;
  }
#pragma omp parallel
{
    #pragma omp for schedule(static,1)
    for(i=0; i<count; i++)
       processwork(parr[i]);
}
```

Count number of items in the linked list

Copy pointer to each node into an array

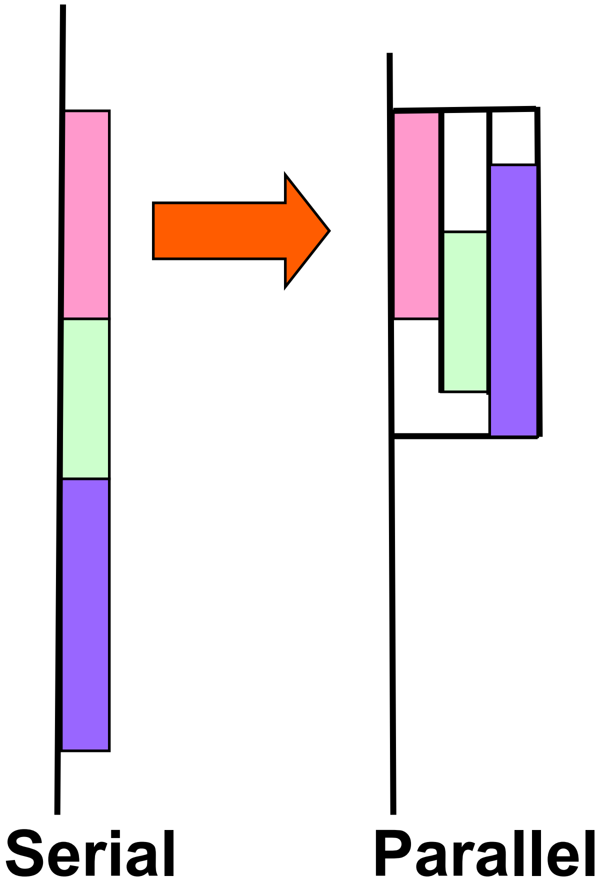
Process nodes in parallel with a for loop

Conclusion

- We were able to parallelize the linked list traversal … but it was ugly and required multiple passes over the data.
- To move beyond its roots in the array based world of scientific computing, we needed to support more general data structures and loops beyond basic for loops.
- To do this, we added tasks in OpenMP 3.0

OpenMP Tasks

- Tasks are independent units of work.
- Tasks are composed of:
  - **code** to execute
  - **data** environment
  - **internal control variables** (ICV)
- Threads perform the work of each task.
- The runtime system decides when tasks are executed
  - Tasks may be deferred
  - Tasks may be executed immediately



Definitions

- *Task construct* – `task` directive plus structured block
- *Task* – the package of code and instructions for allocating data created when a thread encounters a task construct
- *Task region* – the dynamic sequence of instructions produced by the execution of a task by a thread

Intro by Example: Sudoku

Sudoko for Lazy Computer Scientists

- Lets solve Sudoku puzzles with brute multi-core force

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

(1) Search an empty field

(2) Try all numbers:

(2 a) Check Sudoku

(2 b) If invalid:
skip

(2 c) If valid:
Go to next field

Wait for completion

The OpenMP Task Construct

C/C++

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

Fortran

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

- Each encountering thread/task creates a new task
  - → Code and data is being packaged up
  - → Tasks can be nested
    - → Into another task directive
    - → Into a Worksharing construct
- Data scoping clauses:
    - → `shared`(*list*)
    - → `private`(*list*)   `firstprivate`(*list*)
    - → `default`(*shared* | *none*)

Barrier and Taskwait Constructs

- OpenMP `barrier` (implicit or explicit)
  - → All tasks created by any thread of the current *Team* are guaranteed to be completed at barrier exit

C/C++

```
#pragma omp barrier
```

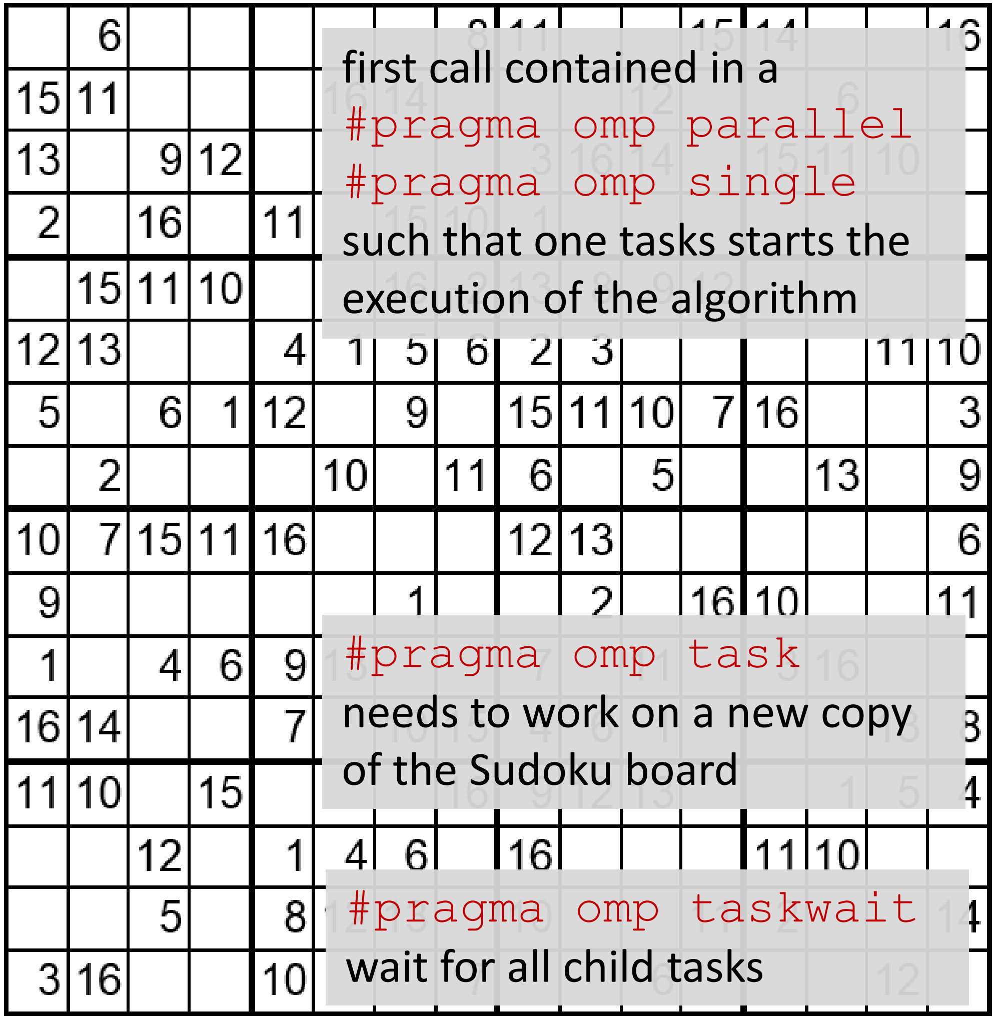
- Task barrier: `taskwait`
  - → Encountering task is suspended until child tasks complete
    - → Applies only to children, not descendants!

C/C++

```
#pragma omp taskwait
```

Parallel Brute-force Sudoku

- This parallel algorithm finds all valid solutions



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

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

`#pragma omp taskwait`
wait for all child tasks

(1) Search an empty field

(2) Try all numbers:

(2 a) Check Sudoku

(2 b) If invalid: skip

(2 c) If valid: Go to next field

Wait for completion

Parallel Brute-force Sudoku (2/3)

- OpenMP parallel region creates a team of threads

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

→Single construct: One thread enters the execution of `solve_parallel`

→the other threads wait at the end of the `single` …

→… and are ready to pick up threads „from the work queue“

- Syntactic sugar (either you like it or you don‘t)

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

Parallel Brute-force Sudoku (3/3)

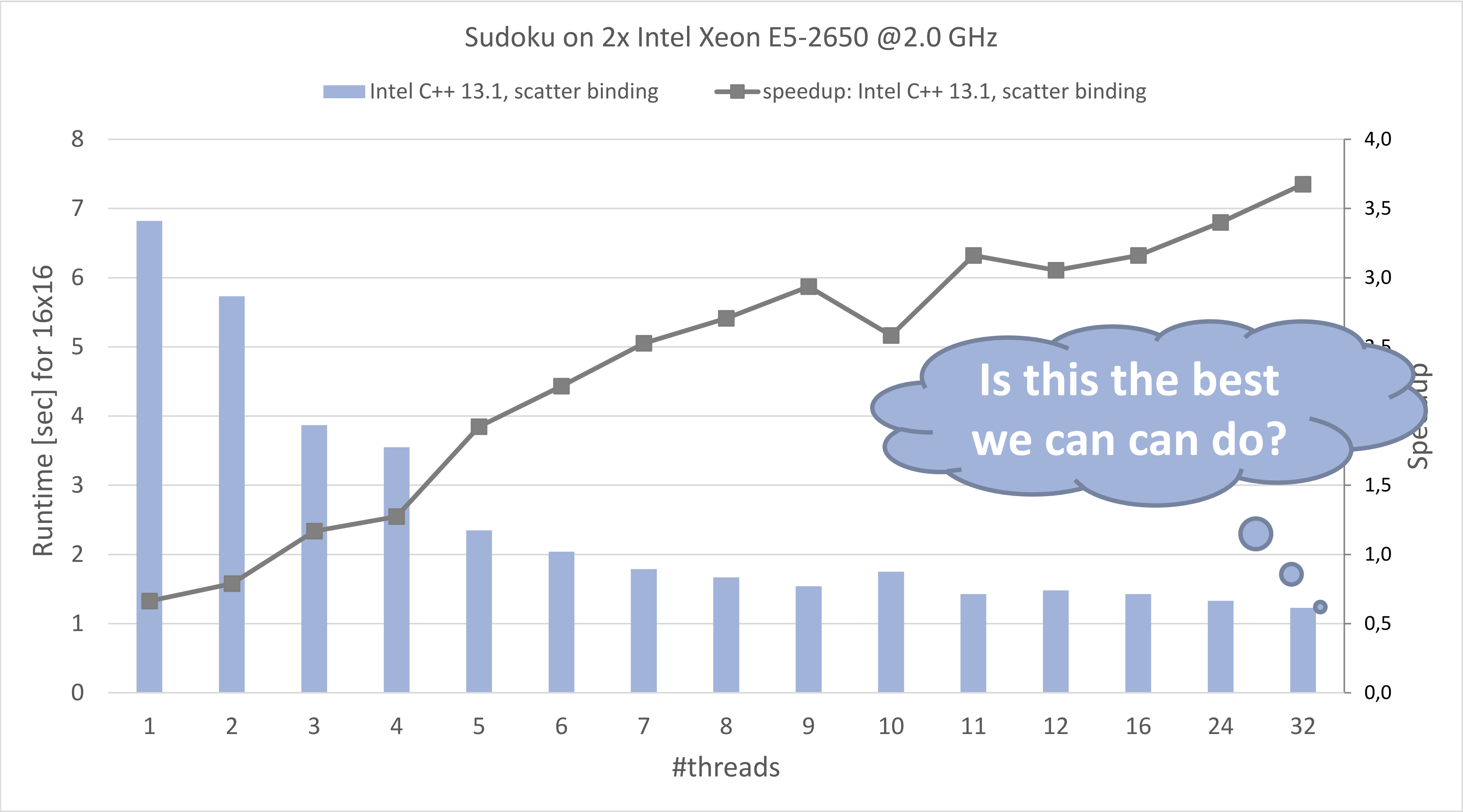
- The actual implementation

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

#pragma omp taskwait
```

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

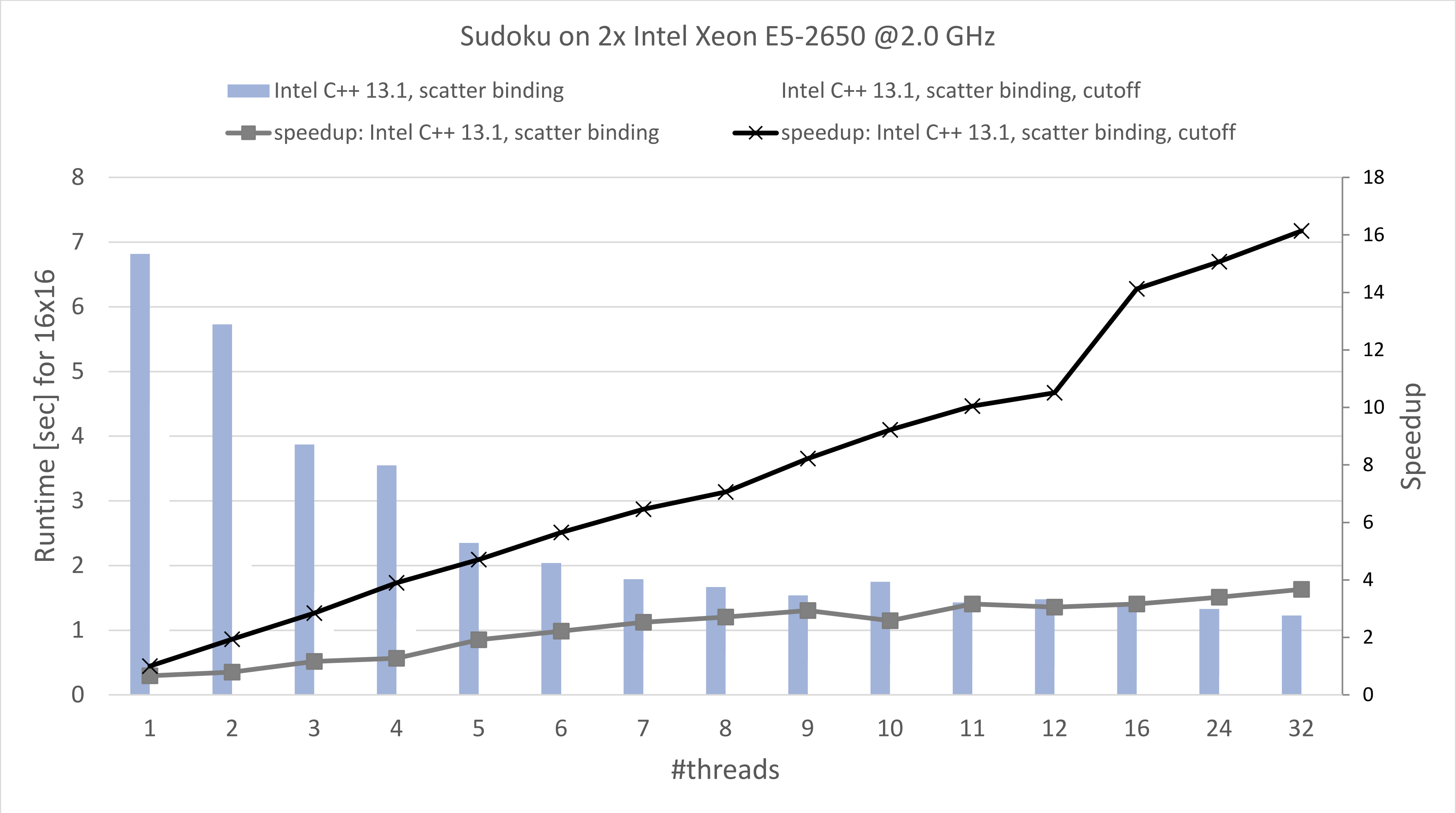
`#pragma omp taskwait`
wait for all child tasks

Performance Evaluation



Is this the best we can can do?

Performance Evaluation



Data Scoping

Tasks in OpenMP: Data Scoping

- Some rules from *Parallel Regions* apply:
  - →Static and Global variables are shared
  - →Automatic Storage (local) variables are private

- If `shared` scoping is not inherited:
  - →Orphaned Task variables are `firstprivate` by default!
  - →Non-Orphaned Task variables inherit the `shared` attribute!
  - → Variables are `firstprivate` unless `shared` in the enclosing context

Data Scoping Example (1/7)

```
int a = 1;
void foo()
{
   int b = 2, c = 3;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d = 4;
        #pragma omp task
        {
               int e = 5;

               // Scope of a:
               // Scope of b:
               // Scope of c:
               // Scope of d:
               // Scope of e:
} } }
```

Data Scoping Example (2/7)

```
int a = 1;
void foo()
{
   int b = 2, c = 3;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d = 4;
        #pragma omp task
        {
               int e = 5;

               // Scope of a: shared
               // Scope of b:
               // Scope of c:
               // Scope of d:
               // Scope of e:
} } }
```

Data Scoping Example (3/7)

```
int a = 1;
void foo()
{
   int b = 2, c = 3;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d = 4;
        #pragma omp task
        {
                int e = 5;

                // Scope of a: shared
                // Scope of b: firstprivate
                // Scope of c:
                // Scope of d:
                // Scope of e:
} } }
```

Data Scoping Example (4/7)

```
int a = 1;
void foo()
{
   int b = 2, c = 3;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d = 4;
        #pragma omp task
        {
                int e = 5;

                // Scope of a: shared
                // Scope of b: firstprivate
                // Scope of c: shared
                // Scope of d:
                // Scope of e:
} } }
```

Data Scoping Example (5/7)

```
int a = 1;
void foo()
{
   int b = 2, c = 3;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d = 4;
        #pragma omp task
        {
                int e = 5;

                // Scope of a: shared
                // Scope of b: firstprivate
                // Scope of c: shared
                // Scope of d: firstprivate
                // Scope of e:
} } }
```

Data Scoping Example (6/7)

```
int a = 1;
void foo()
{
   int b = 2, c = 3;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d = 4;
        #pragma omp task
        {
               int e = 5;

               // Scope of a: shared
               // Scope of b: firstprivate
               // Scope of c: shared
               // Scope of d: firstprivate
               // Scope of e: private
} } }
```

Data Scoping Example (7/7)

```
int a = 1;
void foo()
{
   int b = 2, c = 3;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d = 4;
        #pragma omp task
        {
               int e = 5;

               // Scope of a: shared,        value of a: 1
               // Scope of b: firstprivate,  value of b: 0 / undefined
               // Scope of c: shared,        value of c: 3
               // Scope of d: firstprivate,  value of d: 4
               // Scope of e: private,       value of e: 5
} } }
```

Use default(none)!

Hint: Use default(none) to be forced to think about every variable if you do not see clearly.

```
int a = 1;
void foo()
{
   int b = 2, c = 3;
   #pragma omp parallel shared(b) default(none)
   #pragma omp parallel private(b) default(none)
   {
        int d = 4;
        #pragma omp task
        {
                int e = 5;

                // Scope of a: shared
                // Scope of b: firstprivate
                // Scope of c: shared
                // Scope of d: firstprivate
                // Scope of e: private
} } }
```

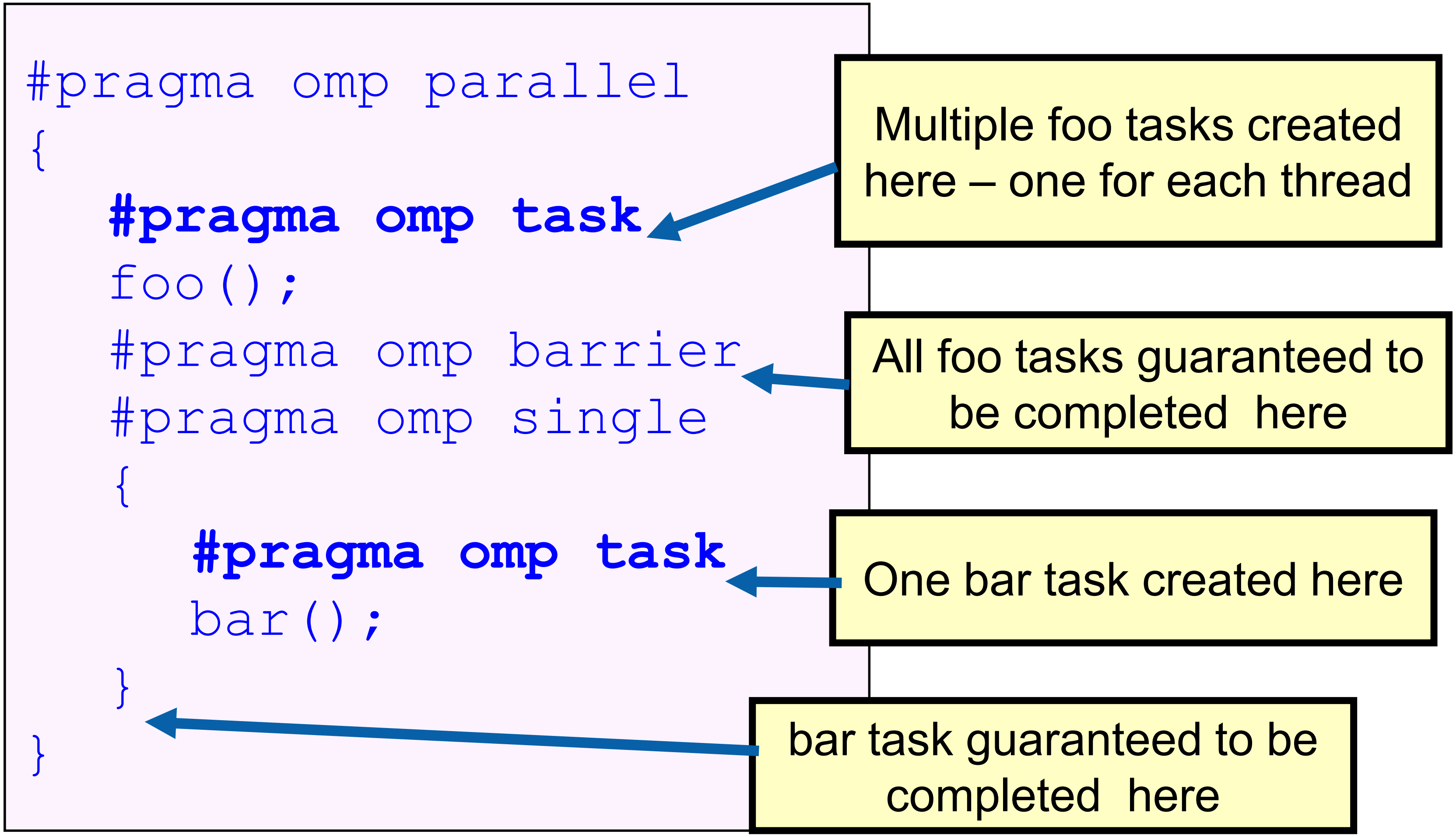
When are tasks guaranteed to complete

- Tasks are guaranteed to be complete at thread barriers:

  #pragma omp barrier

- or task barriers

  #pragma omp taskwait



```
#pragma omp parallel
{
   #pragma omp task
   foo();
   #pragma omp barrier
   #pragma omp single
   {
      #pragma omp task
      bar();
   }
}
```

Multiple foo tasks created here – one for each thread

All foo tasks guaranteed to be completed here

One bar task created here

bar task guaranteed to be completed here

Data Scoping with tasks: Fibonacci example.

This is an instance of the divide and conquer design pattern

```
int fib ( int n )
{

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

n is private in both tasks

x is a private variable
y is a private variable

What’s wrong here?

A task’s private variables are undefined outside the task

Data Scoping with tasks: Fibonacci example.

```
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;
}
```

n is private in both tasks

x & y are shared
Good solution
we need both values to
compute the sum

Data Scoping with tasks: List Traversal example

```
List ml; //my_list
Element *e;
#pragma omp parallel
#pragma omp single
{
   for(e=ml->first;e;e=e->next)
#pragma omp task
      process(e);
}
```

What’s wrong here?

Possible data race !
Shared variable e
updated by multiple tasks

Data Scoping with tasks: List Traversal example

```
List ml; //my_list
Element *e;
#pragma omp parallel
#pragma omp single
{
   for(e=ml->first;e;e=e->next)
#pragma omp task firstprivate(e)
      process(e);
}
```

Good solution – e is firstprivate

Task Construct – Explicit Tasks

```
#pragma omp parallel
{
  #pragma omp single
  {
    node * p = head;
    while (p) {
    #pragma omp task firstprivate(p)
      process(p);
    p = p->next;
    }
  }
}
```

1. Create a team of threads.

2. One thread executes the **single** construct

… other threads wait at the implied barrier at the end of the single construct

3. The “single” thread creates a task with its own value for the pointer p

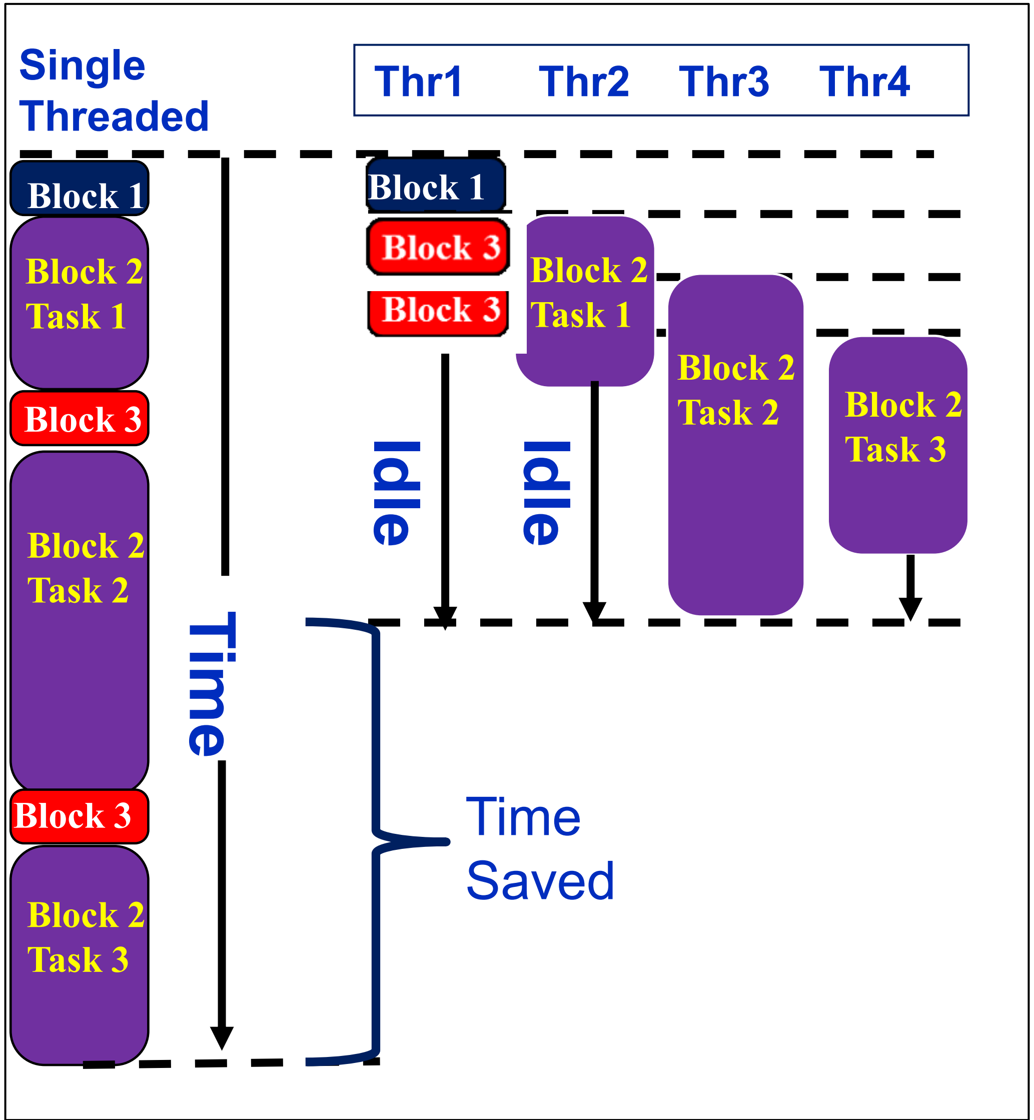
4. Threads waiting at the barrier execute tasks.

Execution moves beyond the barrier once all the tasks are complete

Execution of tasks

Have potential to parallelize irregular patterns and recursive function calls

```
#pragma omp parallel
{
  #pragma omp single
  {   //block 1
    node * p = head;
    while (p) { // block 2
    #pragma omp task
      process(p);
    p = p->next;   //block 3
    }
  }
}
```



Scheduling and Dependencies

Tasks in OpenMP: Scheduling

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

Unsafe use of `untied` Tasks

- Problem: Because untied tasks may migrate between threads at any point, thread-centric constructs can yield unexpected results

- Remember when using `untied` tasks:
  - → Avoid `threadprivate` variable
  - → Avoid any use of thread-ids (i.e., `omp_get_thread_num()`)
  - → Be careful with `critical region` and *locks*

- Possible solution:
  - → Create a tied task region with

```
#pragma omp task if(0)
```

`if` Clause

- If the expression of an `if` clause on a task evaluates to `false`

  →The encountering task is suspended

  →The new task is executed immediately

  →The parent task resumes when the new task finishes

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

The taskyield Directive

- The `taskyield` directive specifies that the current task can be suspended in favor of execution of a different task.
  - →Hint to the runtime for optimization and/or deadlock prevention

C/C++

```
#pragma omp taskyield
```

Fortran

```
!$omp taskyield
```

taskyield Example (1/2)

```
#include <omp.h>

void something_useful();
void something_critical();

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

taskyield Example (2/2)

```
#include <omp.h>

void something_useful();
void something_critical();

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

The waiting task may be suspended here and allow the executing thread to perform other work; may also avoid deadlock situations.

`priority` Clause

C/C++

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

- The *priority* is a hint to the runtime system for task execution order
- Among all tasks ready to be executed, higher priority tasks are recommended to execute before lower priority ones
  - →priority is non-negative numerical scalar (default: 0)
  - →priority <= max-task-priority ICV
    - →environment variable OMP_MAX_TASK_PRIORITY
- It is not allowed to rely on task execution order being determined by this clause!

final Clause

- For recursive problems that perform task decomposition, stopping task creation at a certain depth exposes enough parallelism but reduces overhead.

C/C++

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

Fortran

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

- Merging the data environment may have side-effects

```
void foo(bool arg)
{
    int i = 3;
    #pragma omp task final(arg) firstprivate(i)
        i++;
    printf(“%d\n”, i);   // will print 3 or 4 depending on expr
}
```

mergeable Clause

- If the mergeable clause is present, the implementation might merge the task‘s data environment
  - →if the generated task is undeferred or included
    - →undeferred: if clause present and evaluates to false
    - →included: final clause present and evaluates to true

C/C++

```
#pragma omp task mergeable
```

Fortran

```
!$omp task mergeable
```

- Personal Note: As of today, no compiler or runtime exploits final and/or mergeable so that real world applications would profit from using them ☹.

The taskgroup Construct

C/C++

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

Fortran

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

- Specifies a wait on completion of child tasks and their descendant tasks
  - → „deeper“ sychronization than `taskwait`, but
  - → with the option to restrict to a subset of all tasks (as opposed to a `barrier`)

The depend Clause

C/C++

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

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

Concurrent Execution w/ Dep.

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

- Note: variables in the depend clause do not necessarily have to indicate the data flow

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

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

T2 and T3 can be executed in parallel.

Concurrent Execution w/ Dep.

- The following code allows for more parallelism, as there is one i per thread. Thus, two tasks may be active per thread.

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

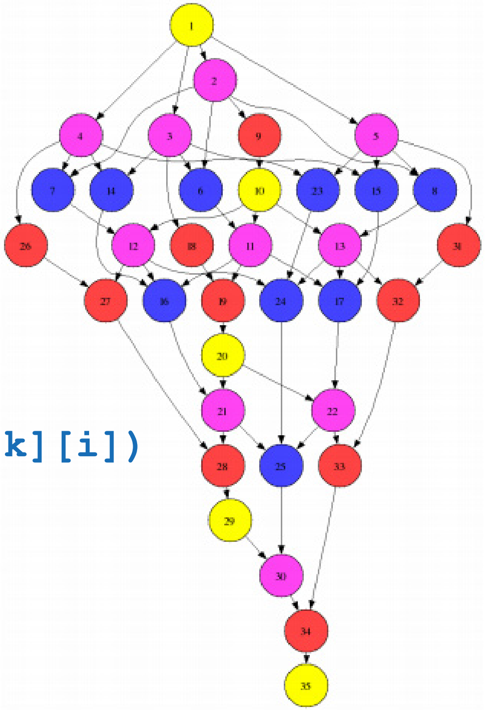
Concurrent Execution w/ Dep.

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

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

„Real“ Task Dependencies

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



* image from BSC

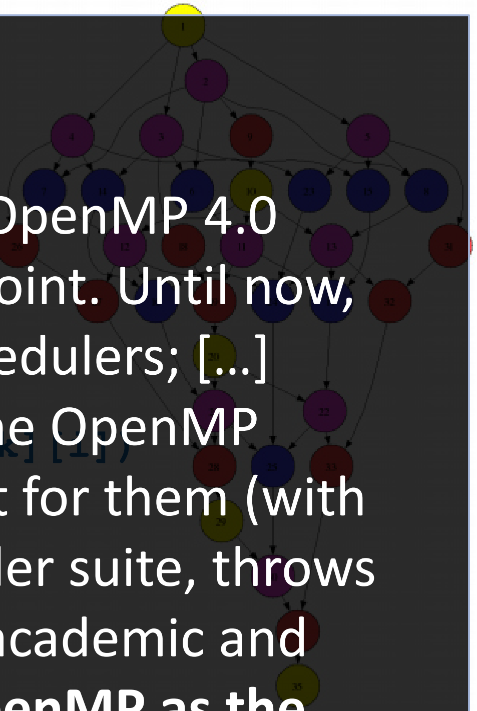
„Real“ Task Dependencies

Jack Dongarra on OpenMP Task Dependencies:

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

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

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



* image from BSC

Vectorization

Topics

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

Auto-vectorization

- Auto vectorization only helps in some cases
  - →Increased complexity of instructions make it hard for the compiler to select highest performance instructions
  - →Code pattern needs to be recognized by the compiler
  - →Precision requirements often inhibit SIMD code gen

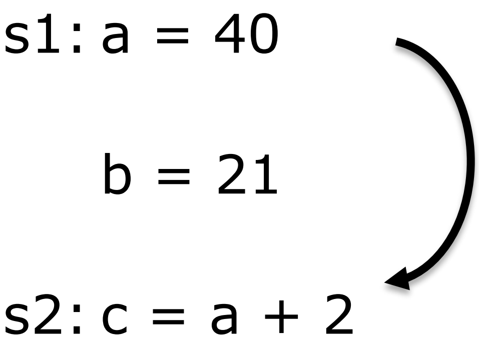
Why Auto-vectorizers Fail

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

Data Dependencies

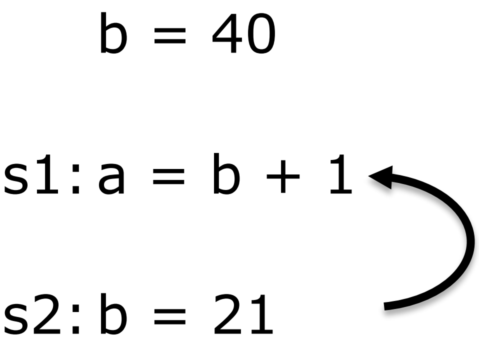
- Suppose two statements S1 and S2
- S2 depends on S1, iff S1 must execute before S2
  - →Control-flow dependence
  - →Data dependence
  - →Dependencies can be carried over between loop iterations
- Important flavors of data dependencies

FLOW



```
s1: a = 40

    b = 21

s2: c = a + 2
```

ANTI



```
    b = 40

s1: a = b + 1

s2: b = 21
```

Loop-Carried Dependencies

- Dependencies may occur across loop iterations
  - →Loop-carried dependency
- The following code contains such a dependency:

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

- Some iterations of the loop have to complete before the next iteration can run

In a Time Before OpenMP 4.0

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

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

You need to trust your compiler to do the “right” thing.

SIMD Loop Construct

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

- Syntax (C/C++)

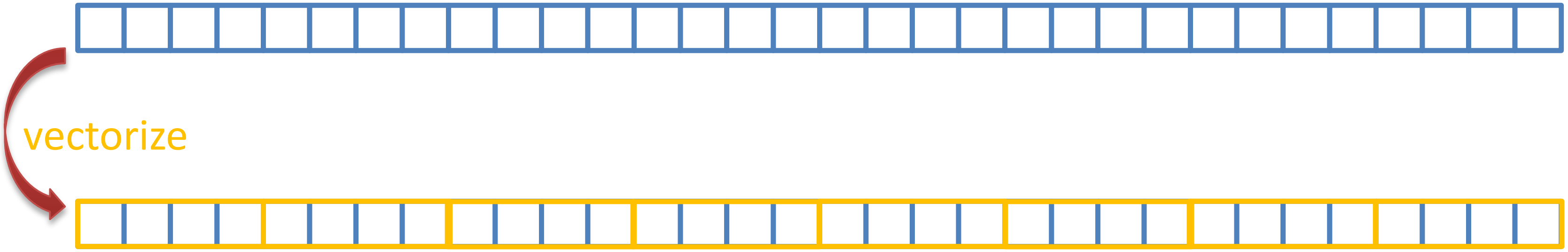
```
#pragma omp simd [clause[[,] clause],…]
for-loops
```

- Syntax (Fortran)

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

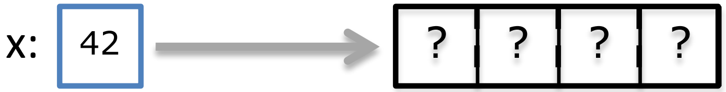
Example

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

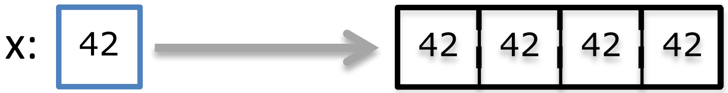


Data Sharing Clauses

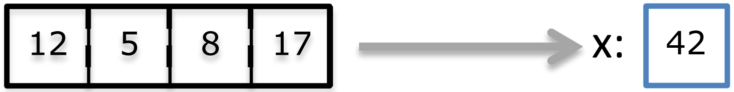
- `private(`*`var-list`*`):`
  Uninitialized vectors for variables in *var-list*


x: 42
? ? ? ?

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


x: 42
42 42 42 42

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


12 5 8 17
x: 42

SIMD Loop Clauses

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

SIMD Worksharing Construct

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

- Syntax (C/C++)

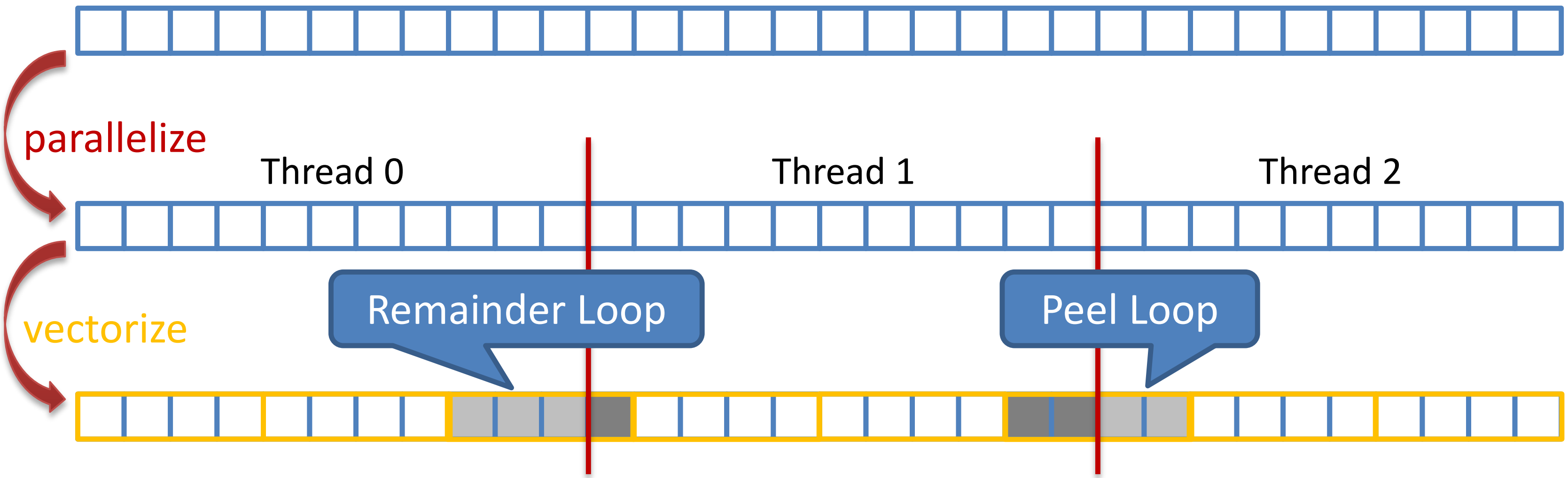
```
#pragma omp for simd [clause[[,] clause],…]
for-loops
```

- Syntax (Fortran)

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

Example

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



Be Careful What You Wish For…

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

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

SIMD Function Vectorization

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


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

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

SIMD Function Vectorization

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

- Syntax (C/C++):

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

- Syntax (Fortran):

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

SIMD Function Vectorization

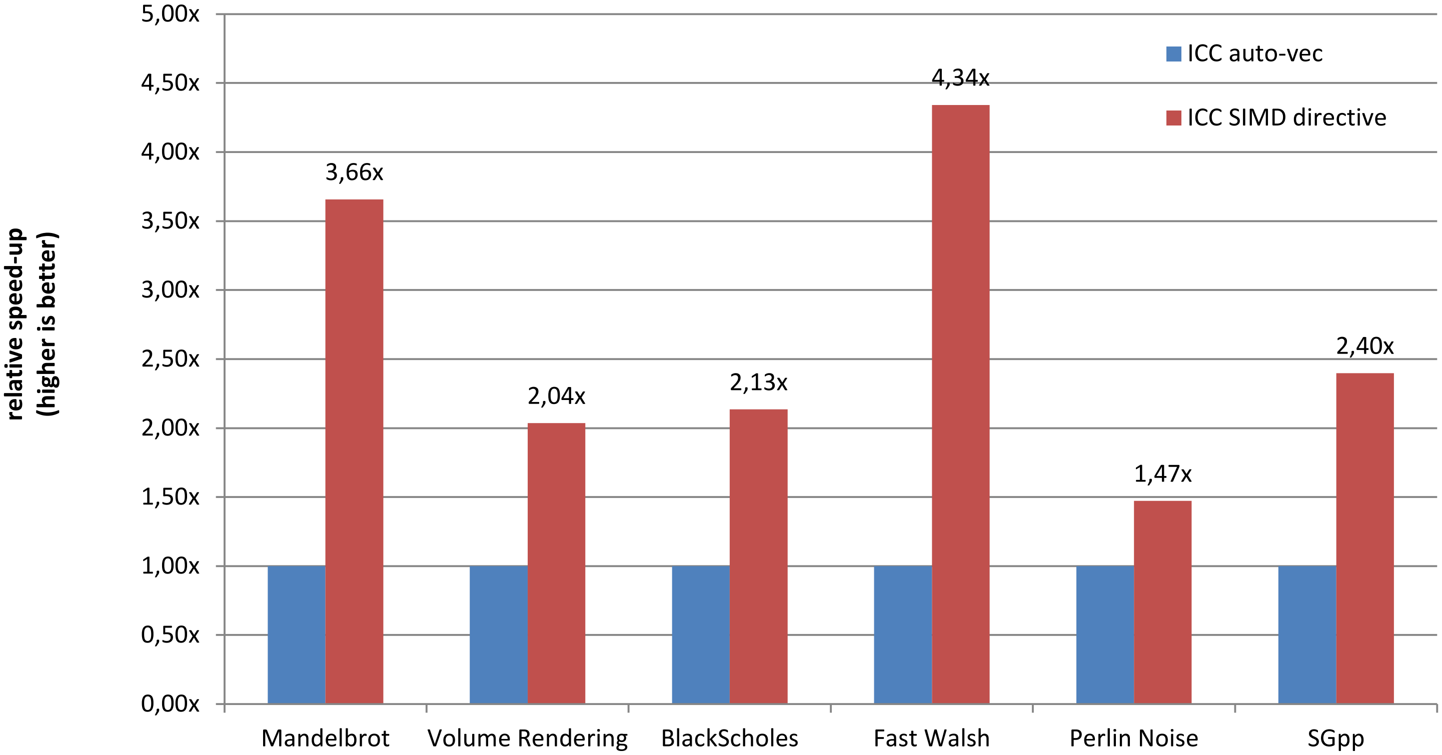
```
#pragma omp declare simd
float min(float a, float b) {
    return a < b ? a : b;
}

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

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

```
vec8 min_v(vec8 a, vec8 b) {
    return a < b ? a : b;
}
```

```
vec8 distsq_v(vec8 x, vec8 y)
    return (x - y) * (x - y);
}
```

```
vd = min_v(distsq_v(va, vb), vc)
```

SIMD Constructs & Performance



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

NUMA

OpenMP and Performance

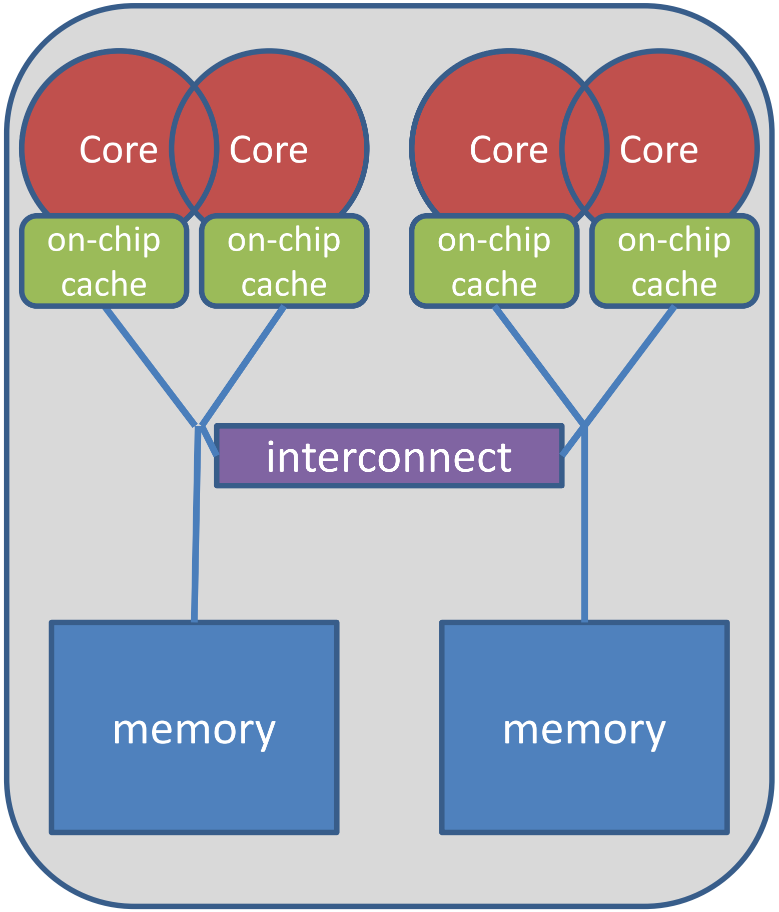
- Two of the more obscure things that can negatively impact performance are cc-NUMA effects and false sharing

- ***Neither of these are restricted to OpenMP***
  - →But they most show up because you used OpenMP
  - →In any case they are important enough to cover here

Non-uniform Memory

How To Distribute The Data ?

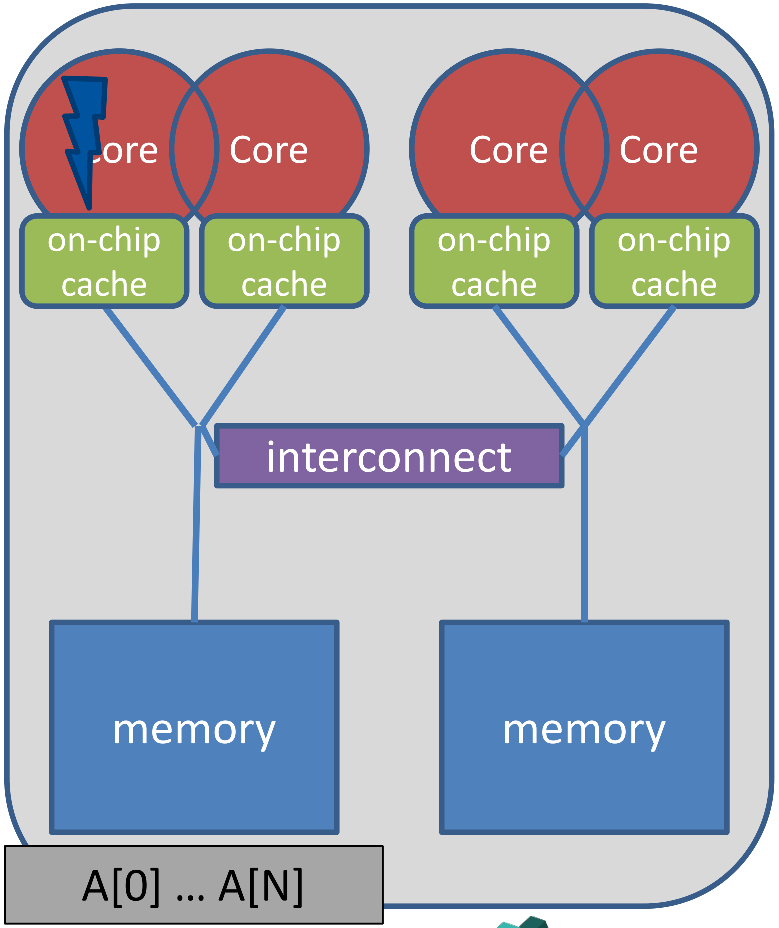
```
double* A;
A = (double*)
    malloc(N * sizeof(double));



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



Non-uniform Memory

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

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




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

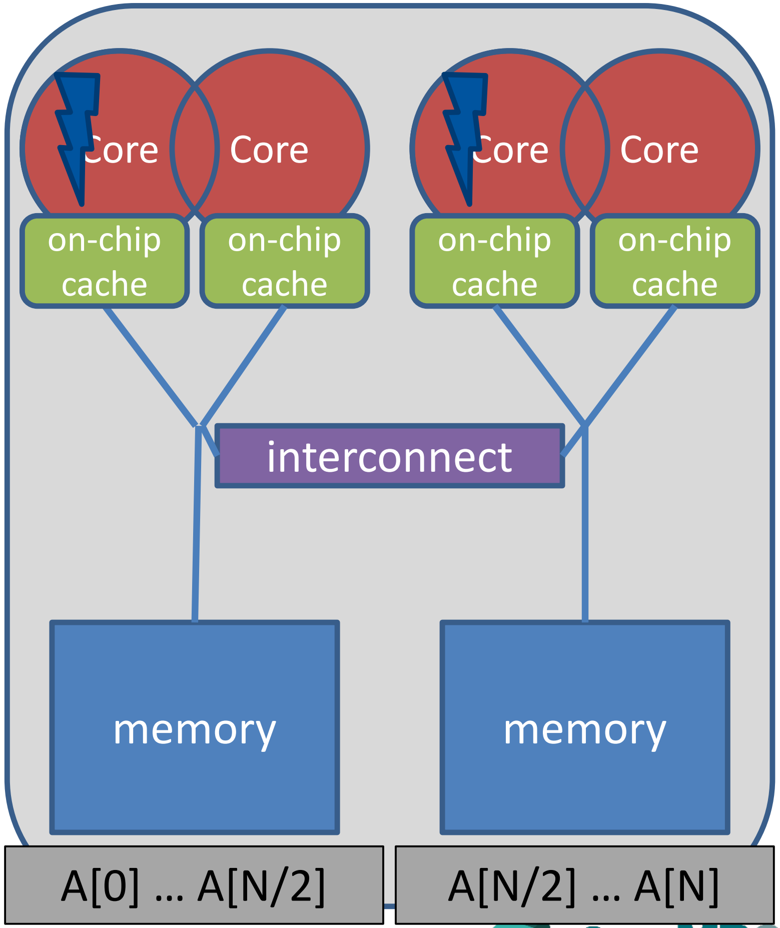


IWOMP 2017

First Touch Memory Placement

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

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

omp_set_num_threads(2);

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

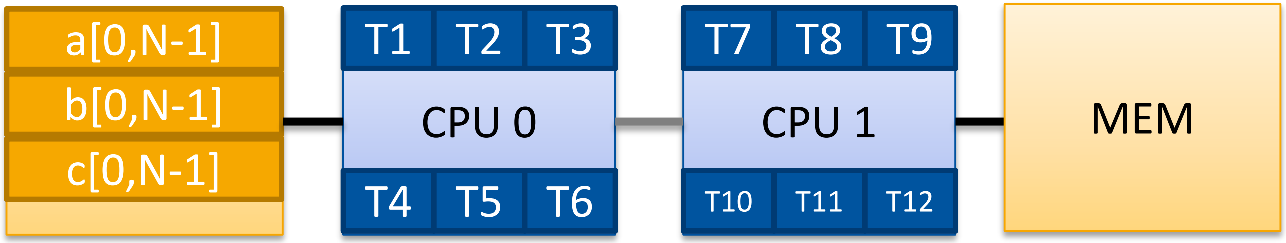


Serial vs. Parallel Initialization

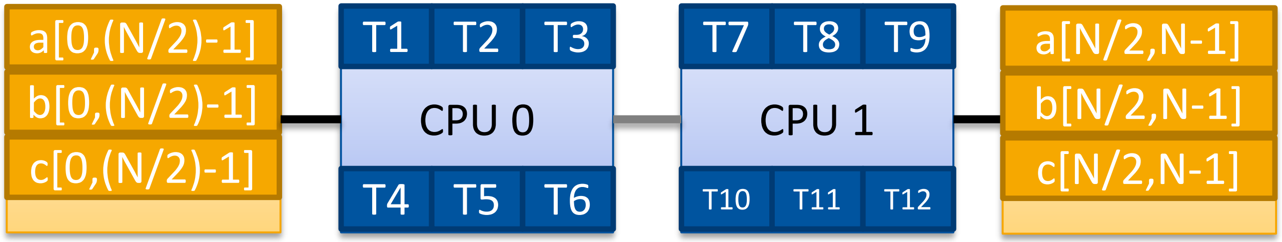
- **Stream example with and without parallel initialization.**
  → 2 socket sytem with Xeon X5675 processors, 12 OpenMP threads

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

ser_init:



par_init:



Thread Binding and Memory Placement

Get Info on the System Topology

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

Decide for Binding Strategy

- **Selecting the „right“ binding strategy depends not only on the topology, but also on the characteristics of your application.**
  - → Putting threads far apart, i.e., on different sockets
    - → May improve the aggregated memory bandwidth available to your application
    - → May improve the combined cache size available to your application
    - → May decrease performance of synchronization constructs
  - → Putting threads close together, i.e., on two adjacent cores that possibly share some caches
    - → May improve performance of synchronization constructs
    - → May decrease the available memory bandwidth and cache size
- **If you are unsure, just try a few options and then select the best one.**

OpenMP 4.0: Places + Policies

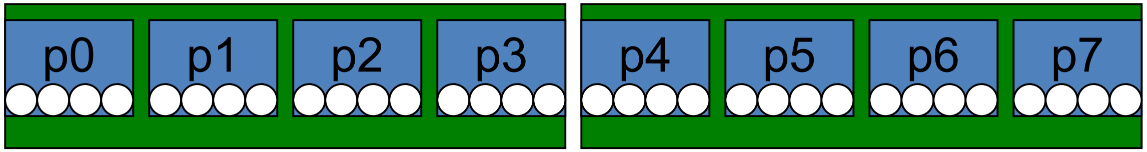
- **Define OpenMP places**
  - → set of OpenMP threads running on one or more processors
  - → can be defined by the user, i.e., `OMP_PLACES=cores`

- **Define a set of OpenMP thread affinity policies**
  - → SPREAD: spread OpenMP threads evenly among the places, partition the place list
  - → CLOSE: pack OpenMP threads near master thread
  - → MASTER: collocate OpenMP thread with master thread

- **Goals**
  - → user has a way to specify where to execute OpenMP threads for locality between OpenMP threads / less false sharing / memory bandwidth

OMP_PLACES env. variable

- Assume the following machine:



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

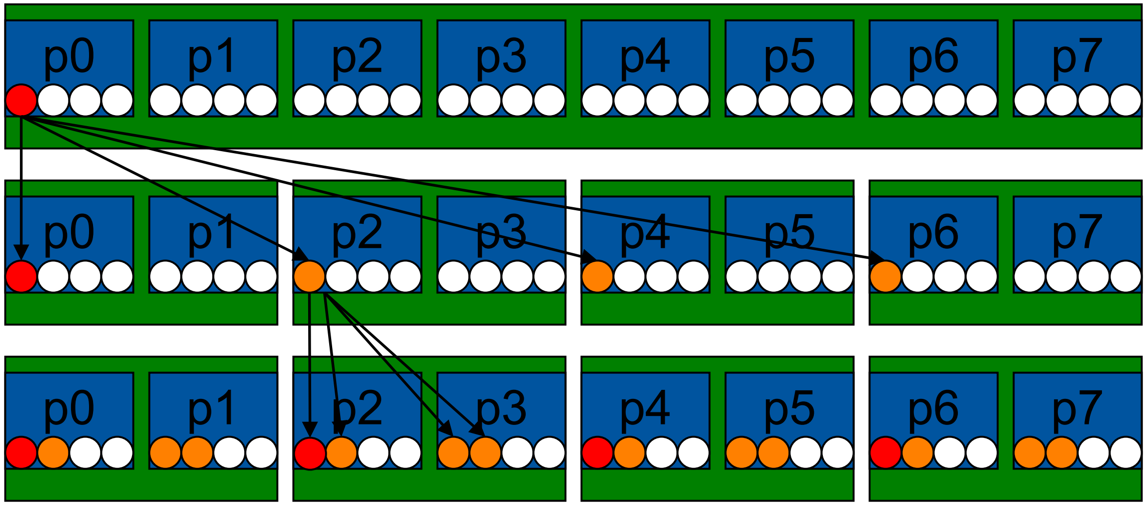
- Abstract names for OMP_PLACES:
  → threads: Each place corresponds to a single hardware thread on the target machine.
  → cores: Each place corresponds to a single core (having one or more hardware threads) on the target machine.
  → sockets: Each place corresponds to a single socket (consisting of one or more cores) on the target machine.

OpenMP 4.0: Places + Binding Policies (2/2)

- **Example‘s Objective:**
  - → separate cores for outer loop and near cores for inner loop
- **Outer Parallel Region: proc_bind(spread), Inner: proc_bind(close)**
  - → spread creates partition, compact binds threads within respective partition

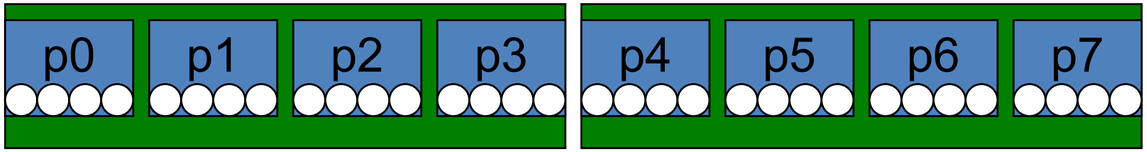
```
OMP_PLACES=(0,1,2,3), (4,5,6,7), ... = (0-3):8:4    = cores
#pragma omp parallel proc_bind(spread) num_threads(4)
#pragma omp parallel proc_bind(close) num_threads(4)
```

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



More Examples (1/3)

- Assume the following machine:



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

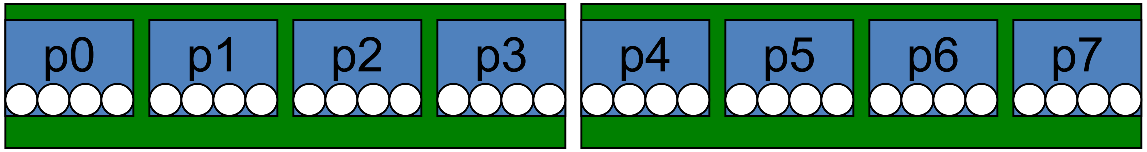
- Parallel Region with two threads, one per socket

→`OMP_PLACES=sockets`

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

More Examples (2/3)

- Assume the following machine:



- Parallel Region with four threads, one per core, but only on the first socket
  - →`OMP_PLACES=cores`
  - →`#pragma omp parallel num_threads(4) \`
    `proc_bind(close)`

More Examples (3/3)

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

Places API (1/2)

- 1: Query information about binding and a single place of all places with ids 0 … `omp_get_num_places()`:
- `omp_proc_bind_t omp_get_proc_bind()`: returns the thread affinity policy (omp_proc_bind_false, true, master, …)
- `int omp_get_num_places()`: returns the number of places
- `int omp_get_place_num_procs(int place_num)`: returns the number of processors in the given place
- `void omp_get_place_proc_ids(int place_num, int* ids)`: returns the ids of the processors in the given place

Places API (2/2)

- 2: Query information about the place partition:
- `int omp_get_place_num()`: returns the place number of the place to which the current thread is bound
- `int omp_get_partition_num_places()`: returns the number of places in the current partition
- `void omp_get_partition_place_nums(int* pns)`: returns the list of place numbers corresponding to the places in the current partition

Places API: Example

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

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

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

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

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

     free(place_processors);
}
```

A First Summary

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

- What if the data access pattern changes over time?

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

NUMA Strategies: Overview

- First Touch: Modern operating systems (i.e., Linux >= 2.4) determine the physical location of a memory page during the first page fault, when the page is first „touched“, and put it close to the CPU that causes the page fault

- Explicit Migration: Selected regions of memory (pages) are moved from one NUMA node to another via explicit OS syscall

- Next Touch: The binding of pages to NUMA nodes is removed and pages are put in the location of the next „touch“; well supported in Solaris, expensive to implement in Linux

- Automatic Migration: No support for this in current operating systems

User Control of Memory Affinity

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

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

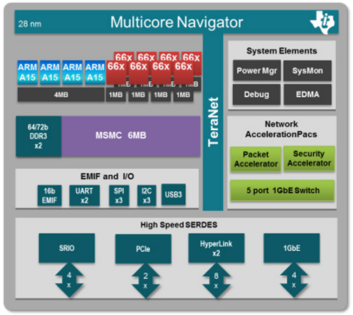
Offload Programming

Topics

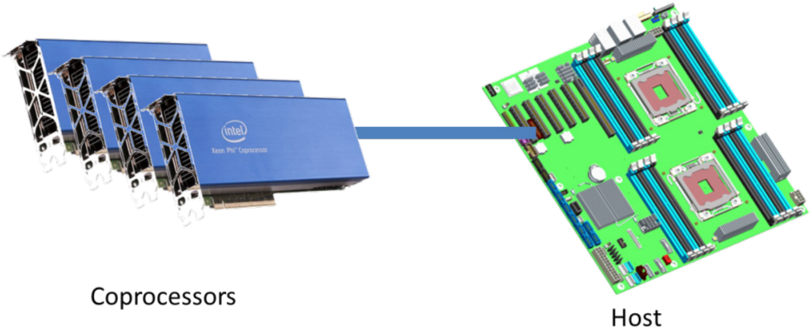
- Heterogeneous device execution model
- Mapping variables to a device
- Accelerated workshare

Device Model

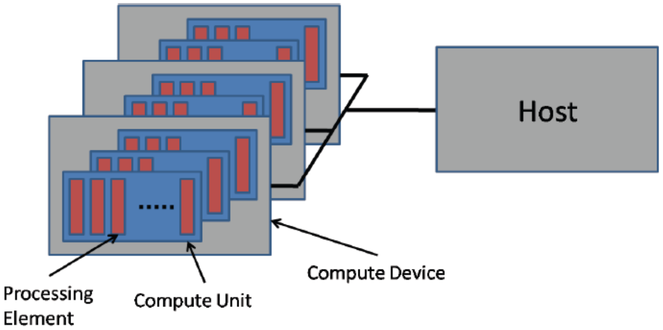
- OpenMP 4.0 supports heterogeneous systems
- Device model:
  - →One host device and
  - →One or more target devices



Heterogeneous SoC



Host and Co-processors



Host and GPUs

Terminology

- Device:
  an implementation-defined (logical) execution unit
- Device data environment:
  The storage associated with a device.

The execution model is host-centric such that the host device offloads **target** regions to target devices.

OpenMP 4.0 Device Constructs

- Execute code on a target device
  - **omp target** *[clause[[,] clause],…]*
    structured-block
  - **omp declare target**
    [function-definitions-or-declarations]
- Map variables to a target device
  - **map** *([map-type:] list) // map clause*
    map-type := ***alloc*** | ***tofrom*** | ***to*** | ***from***
  - **omp target data** *[clause[[,] clause],…]*
    structured-block
  - **omp target update** *[clause[[,] clause],…]*
  - **omp declare target**
    [variable-definitions-or-declarations]
- Workshare for acceleration
  - **omp teams** *[clause[[,] clause],…]*
    structured-block
  - **omp distribute** *[clause[[,] clause],…]*
    for-loops

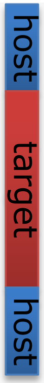
Device Runtime Support

- Runtime support routines
  - `void omp_set_default_device(int dev_num )`
  - `int omp_get_default_device(void)`
  - `int omp_get_num_devices(void);`
  - `int omp_get_num_teams(void)`
  - `int omp_get_team_num(void);`
  - `Int omp_is_initial_device(void);`
- Environment variable
  - Control default device through **`OMP_DEFAULT_DEVICE`**
  - Accepts a non-negative integer value

Offloading Computation

- Use target construct to
  - →Transfer control from the host to the target device
  - →Map variables between the host and target device data environments
- Host thread waits until offloaded region completed
  - →Use other OpenMP tasks for asynchronous execution

```
#pragma omp target map(to:b,c,d) map(from:a)
  {
#pragma omp parallel for
    for (i=0; i<count; i++) {
      a[i] = b[i] * c + d;
    }
  }
```



`target` Construct

- Transfer control from the host to the device
- Syntax (C/C++)

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

- Syntax (Fortran)

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

- Clauses

```
device(scalar-integer-expression)
map(alloc | to | from | tofrom: list)
if(scalar-expr)
```

`map` Clause

```c
extern void init(float*, float*, int);
extern void output(float*, int);

void vec_mult(float *p, float *v1, float *v2, int N)
{
   int i;
   init(v1, v2, N);

   #pragma omp target map(v1[0:N], v2[:N], p[0:N])
   #pragma omp parallel for
   for (i=0; i<N; i++)
     p[i] = v1[i] * v2[i];

   output(p, N);
}
```

```fortran
module mults
contains
subroutine vec_mult(p,v1,v2,N)
   real,dimension(*) :: p, v1, v2
   integer :: N,i
   call init(v1, v2, N)
   !$omp target map(v1(1:N), v2(:N), p(1:N))
   !$omp parallel do
   do i=1,N
      p(i) = v1(i) * v2(i)
   end do
   !omp end target
   call output(p, N)
end subroutine
end module
```

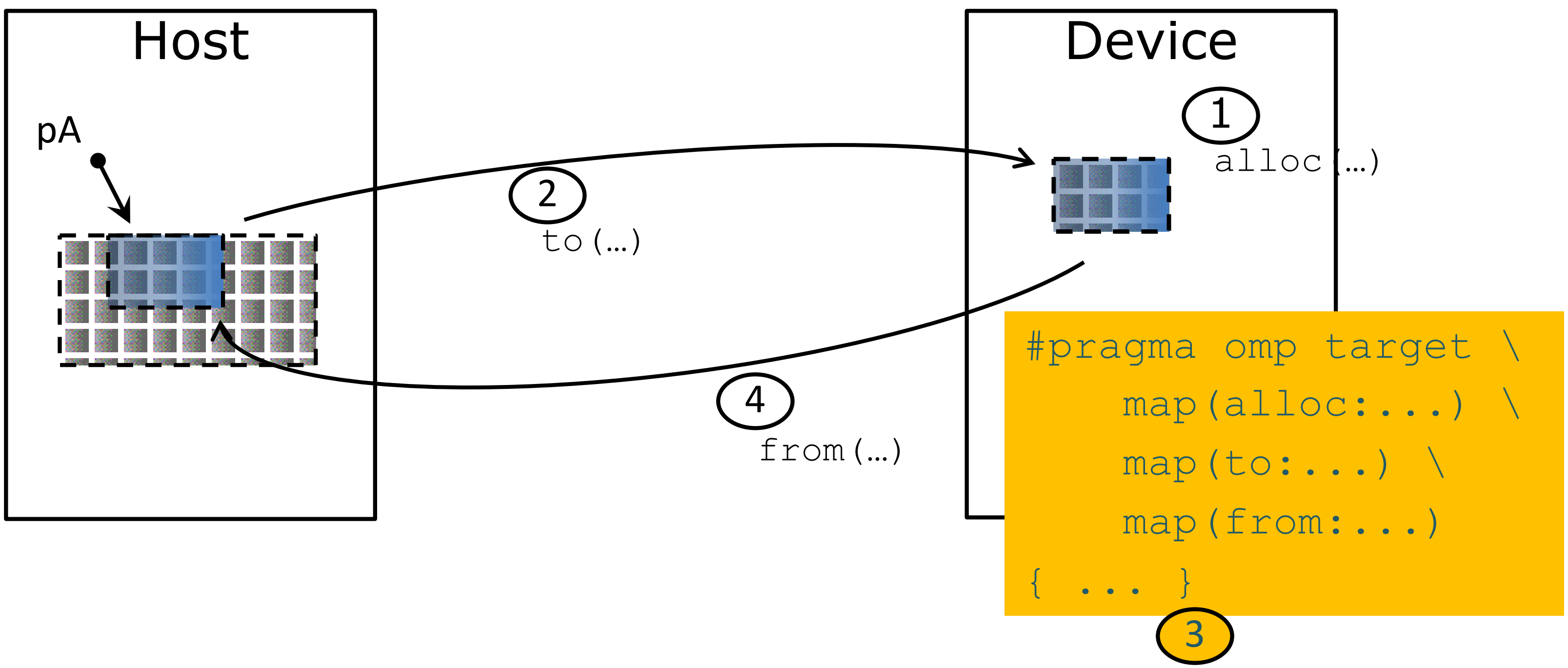
- The array sections for v1, v2, and p are explicitly *mapped* into the device data environment.
- The variable N is implicitly *mapped* into the device data environment

Terminology

- Mapped variable:
  An *original variable* in a (host) data environment has a *corresponding variable* in a device data environment
- Mappable type:
  A type that is amenable for mapped variables.
  (Bitwise copyable plus additional restrictions.)

Device Data Environment

- The **`map`** clauses determine how an *original variable* in a data environment is mapped to a *corresponding variable* in a device data environment.



```
#pragma omp target \
    map(alloc:...) \
    map(to:...) \
    map(from:...)
{ ... }
```

`map` Clause

- Map a variable or an array section to a device data environment
- Syntax:

```
map([map-type:] list)
```

- Where map-type is:
  - → `alloc:` allocate storage for corresponding variable
  - → `to:` alloc and assign value of original variable to corresponding variable on entry
  - → `from:` alloc and assign value of corresponding variable to original variable on exit
  - → `tofrom:` default, both to and form

map Clause

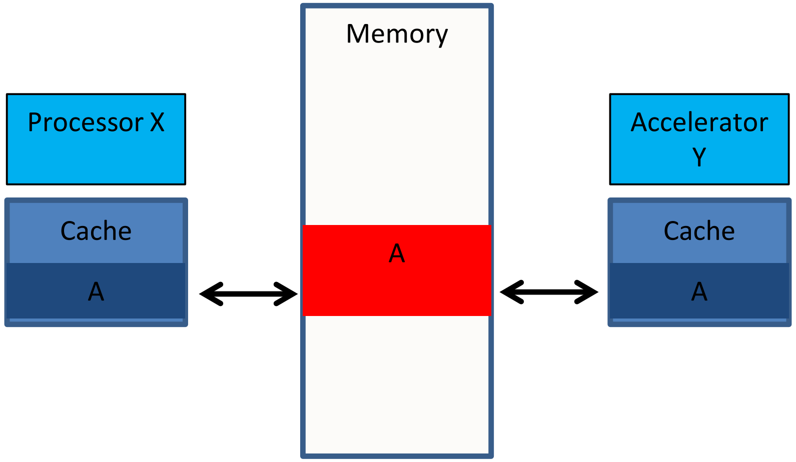
```
extern void init(float*, float*, int);
extern void output(float*, int);

void vec_mult(float *p, float *v1, float *v2, int N)
{
   int i;
   init(v1, v2, N);

   #pragma omp target map(to:v1[0:N],v2[:N]) map(from:p[0:N])
   #pragma omp parallel for
   for (i=0; i<N; i++)
     p[i] = v1[i] * v2[i];

   output(p, N);
}
```

```
module mults
contains
subroutine vec_mult(p,v1,v2,N)
   real,dimension(*) :: p, v1, v2
   integer :: N,i
   call init(v1, v2, N)
   !$omp target map(to: v1(1:N), v2(:N)) map(from:p(1:N))
   !$omp parallel do
   do i=1,N
      p(i) = v1(i) * v2(i)
   end do
   !omp end target
   call output(p, N)
end subroutine
end module
```

- On entry to the target region:
  - Allocate corresponding variables v1, v2, and p in the device data environment.
  - Assign the corresponding variables v1 and v2 the value of their respective original variables.
  - The corresponding variable p is undefined.
- On exit from the target region:
  - Assign the original variable p the value of its corresponding variable.
  - The original variables v1 and v2 are undefined.
  - Remove the corresponding variables v1, v2, and p from the device data environment

MAP is not necessarily a copy

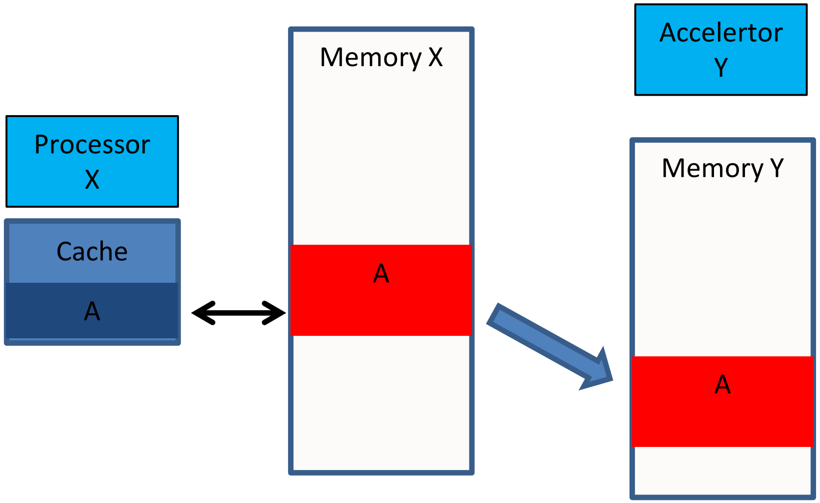
Shared memory



- The corresponding variable in the device data environment *may* share storage with the original variable.

- Writes to the corresponding variable may alter the value of the original variable.

Distributed memory



Map variables across multiple target regions

- Optimize sharing data between host and device
- The `target data` construct maps variables but does not offload code.
- Corresponding variables remain in the device data environment for the extent of the target data region
- Useful to map variables across multiple target regions
- The `target update` synchronizes an original variable with its corresponding variable.

target data Construct Example

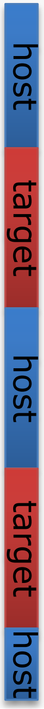
```
extern void init(float*, float*, int);
extern void init_again(float*, float*, int);
extern void output(float*, int);

void vec_mult(float *p, float *v1, float *v2, int N)
{
   int i;

   init(v1, v2, N);

   #pragma omp target data map(from: p[0:N])
   {
      #pragma omp target map(to: v1[:N], v2[:N])
      #pragma omp parallel for
      for (i=0; i<N; i++)
          p[i] = v1[i] * v2[i];

      init_again(v1, v2, N);

      #pragma omp target map(to: v1[:N], v2[:N])
      #pragma omp parallel for
      for (i=0; i<N; i++)
          p[i] = p[i] + (v1[i] * v2[i]);
   }

   output(p, N);
}
```

- The `target data` construct maps variables to the *device data environment.*
- v1 and v2 are mapped at each `target` construct.
- p is mapped once by the `target data` construct.

Map variables to a device data environment

- The host thread executes the data region
- Be careful when using the device clause

```
#pragma omp target data device(0) map(alloc:tmp[:N]) map(to:input[:N)) map(from:res)
  {
#pragma omp target device(0)
#pragma omp parallel for
    for (i=0; i<N; i++)
      tmp[i] = some_computation(input[i], i);

    do_some_other_stuff_on_host();

#pragma omp target device(0)
#pragma omp parallel for reduction(+:res)
    for (i=0; i<N; i++)
      res += final_computation(tmp[i], i)
  }
```



`target data` Construct

- Map variables to a device data environment for the extent of the region.
- Syntax (C/C++)

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

- Syntax (Fortran)

```
!$omp target data [clause[[,] clause],…]
structured-block
!$omp end target data
```

- Clauses

```
device(scalar-integer-expression)
map(alloc | to | from | tofrom: list)
if(scalar-expr)
```

Synchronize mapped variables

- Synchronize the value of an original variable in a host data environment with a corresponding variable in a device data environment

```
#pragma omp target data map(alloc:tmp[:N]) map(to:input[:N)) map(from:res)
  {
#pragma omp target
#pragma omp parallel for
    for (i=0; i<N; i++)
      tmp[i] = some_computation(input[i], i);

    update_input_array_on_the_host(input);

#pragma omp target update device(0) to(input[:N])

#pragma omp target
#pragma omp parallel for reduction(+:res)
    for (i=0; i<N; i++)
      res += final_computation(input[i], tmp[i], i)
  }
```

host

target

host

target

host

`target update` Construct

- Issue data transfers between host and devices
- Syntax (C/C++)

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

- Syntax (Fortran)

  ```
  !$omp target update [clause[[,] clause],…]
  ```

- Clauses

  ```
  device(scalar-integer-expression)
  to(list)
  from(list)
  if(scalar-expr)
  ```

Map a variable for the whole program

```
define N 1000
#pragma omp declare target
float p[N], v1[N], v2[N];
#pragma omp end declare target

extern void init(float *, float *, int);
extern void output(float *, int);

void vec_mult()
{
   int i;
   init(v1, v2, N);
   #pragma omp target update to(v1, v2)
   #pragma omp target
   #pragma omp parallel for
   for (i=0; i<N; i++)
      p[i] = v1[i] * v2[i];

   #pragma omp target update from(p)
   output(p, N);
}
```

- Indicate that global variables are mapped to a device data environment for the whole program
- Use `target update` to maintain consistency between host and device

`target declare` Construct

- Map variables to a device data environment for the whole program
- Syntax (C/C++):

```
#pragma omp declare target
    [variable-definitions-or-declarations]
#pragma omp end declare target
```

- Syntax (Fortran):

```
!$omp declare target (list)
```

Call functions in a target region

- Function declaration must appear in a declare target construct
- The functions will be compiled for
  - → Host execution (as usual)
  - → Target execution (to be invoked from target regions)

```
#pragma omp declare target
float some_computation(float fl, int in) {
  // ... code ...
}

float final_computation(float fl, int in) {
  // ... code ...
}
#pragma omp end declare target
```

Terminology

- League:
  the set of threads teams created by a `teams` construct
- Contention group:
  threads of a team in a league and their descendant threads

`teams` Construct

- The **`teams`** construct creates a *league* of thread teams
  - The master thread of each team executes the **`teams`** 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

`teams` Construct – Restrictions

- A `teams` constructs must be "perfectly" nested in a `target` construct:
  - → No statements or directives outside the `teams` construct
- Only special OpenMP constructs can be nested inside a `teams` construct:
  - → `distribute` (see next slides)
  - → `parallel`
  - → `parallel for` (C/C++), `parallel do` (Fortran)
  - → `parallel sections`

`teams` Construct

- Syntax (C/C++):

```
#pragma omp teams [clause[[,] clause],…]
structured-block
```

- Syntax (Fortran):

```
!$omp teams [clause[[,] clause],…]
structured-block
```

- Clauses

```
num_teams(integer-expression)
thread_limit(integer-expression)
default(shared | none)
private(list), firstprivate(list)
shared(list), reduction(operator : list)
```

`distribute` Construct

- New kind of worksharing construct
  - → Distribute the iterations of the associated loops across the master threads of each team executing the region
  - → No implicit barrier at the end of the construct

- `dist_schedule(`*`kind[, chunk_size]`*`)`
  - → If specified scheduling kind must be static
  - → Chunks are distributed in round-robin fashion of chunks with size *`chunk_size`*
  - → If no chunk size specified, chunks are of (almost) equal size; each team receives at least one chunk

`distribute` Construct

- Syntax (C/C++):

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

- Syntax (Fortran):

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

- Clauses

```
private(list)
firstprivate(list)
collapse(n)
dist_schedule(kind[, chunk_size])
```

SAXPY: on device

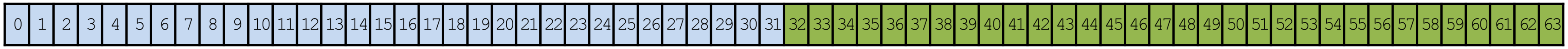
```
void saxpy(float * restrict y, float * restrict x, float a, int n)
{
#pragma omp target map(to:n,a,x[:n]) map(y[:n])
  {
#pragma omp parallel for
   for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
  }
  }
}
```

- How to run this loop in parallel on massively parallel hardware which typically has many clusters of execution units (or cores)?
- Chunk loop level 1: distribute big chunks of loop iterations to each cluster (thread blocks, coherency domains, card, etc...) – to each team
- Chunk loop level 2: loop workshare the iterations in a distributed chunk across threads in a team.
- Chunk loop level 3: Use simd level parallelism inside each thread.

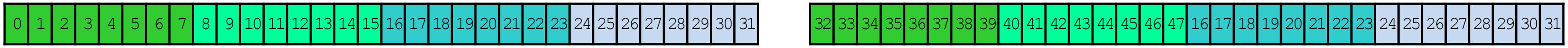
Distribute Parallel SIMD

64 iterations assigned to 2 teams; Each team has 4 threads; Each thread has 2 simd lanes

Distribute Iterations across 2 teams

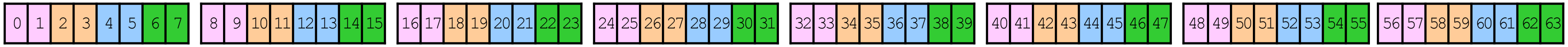

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

In a team workshare iterations across 4 threads


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

In a thread use simd parallelism


0 1 2 3 4 5 6 7

8 9 10 11 12 13 14 15

16 17 18 19 20 21 22 23

24 25 26 27 28 29 30 31

32 33 34 35 36 37 38 39

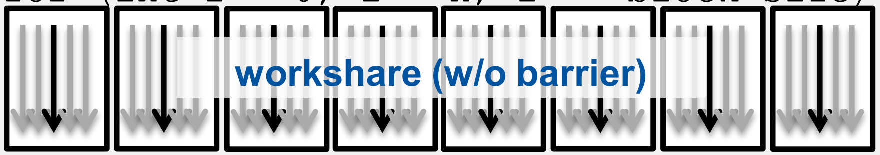
40 41 42 43 44 45 46 47

48 49 50 51 52 53 54 55

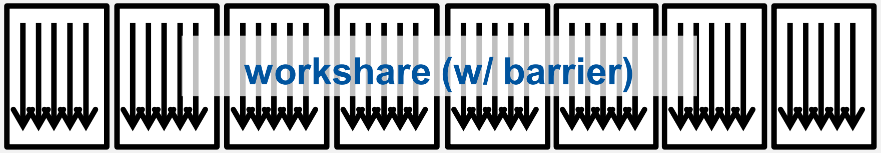
56 57 58 59 60 61 62 63

SAXPY: Accelerated worksharing

```
void saxpy(float * restrict y, float * restrict x, float a, int n)
{
#pragma omp target teams map(to:n,a,x[:n]) map(y[:n])
{
 int block_size = n/omp_get_num_teams();

#pragma omp distribute dist_sched(static, 1)
  for (int i = 0; i < n; i += block_size){
```



```
#pragma omp parallel for
    for (int j = i; j < i + block_size; j++) {
```



```
        y[j] = a*x[j] + y[j];
}}
}
```

Combined Constructs

- The distribution patterns can be cumbersome

- OpenMP 4.0 defines composite constructs for typical code patterns
  - `distribute simd`
  - `distribute parallel for` (C/C++)
  - `distribute parallel for simd` (C/C++)
  - `distribute parallel do` (Fortran)
  - `distribute parallel do simd` (Fortran)
  - ... plus additional combinations for `teams` and `target`
- Avoids the need to do manual loop blocking

SAXPY: Combined Constructs

```
void saxpy(float * restrict y, float * restrict x, float a, int n)
{

#pragma omp target map(to:n,a,x[:n]) map(y[:n])
#pragma omp teams distribute parallel for
for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}
```

```
void saxpy(float * restrict y, float * restrict x, float a, int n)
{

#pragma omp target map(to:n,a,x[:n]) map(y[:n])
#pragma omp teams distribute
for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}
```

Comparing OpenMP with OpenACC

OpenMP 4.0 – accelerated workshare

```
#pragma omp target teams map(B[0:N]) num_teams(numblocks)
#pragma omp distribute parallel for
for (i=0; i<N; ++1) {
  B[i] += sin(B[i]);
}
```

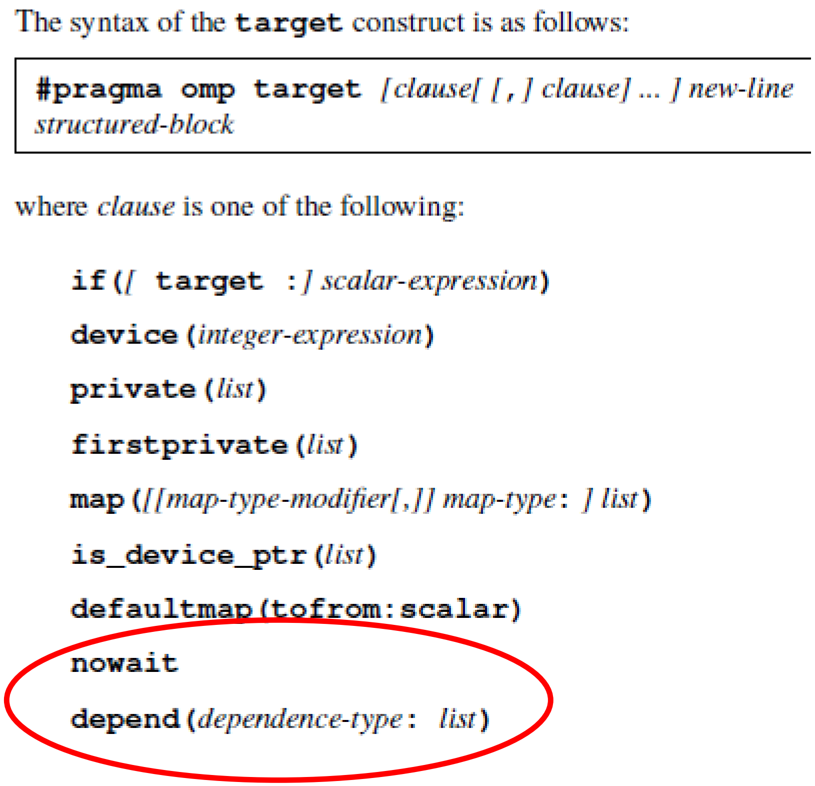
OpenACC – accelerated workshare

```
#pragma acc parallel copy(B[0:N]) num_gangs(numblocks)
#pragma acc loop gang worker
  for (i=0; i<N; ++i) {
    B[i] += sin(B[i]);
  }
```

New in OpenMP 4.5

- Unstructured Data Mapping
  - → **`target enter data map(`*`list`*`)`**
  - → **`target exit data map(`*`list`*`)`**
- Asynchronous Execution
  - → target regions are now task regions
  - → **`target depend(in: x) nowait`**
- Performance oriented changes
  - → Scalar variables are firstprivate
  - → Improvements for Array sections in C/C++
- New device Memory API routines
  - → **`omp_target_free()`**, **`omp_target_alloc()`**, etc…
- Support for device pointers
  - → **`target data use_deviceptr(a)`**
  - → **`target is_deviceptr(p)`**
- Mapping structure elements
  - → **`target map(S.x, s.y[:N])`**
- New combined constructs
  - → **`target parallel`** …
- New ways to map global variables
  - → **`declare target link(x)`**
- Extensions to the if clause for combined/composite constructs
  - → **`target parallel if(target,c1) if(parallel,c2)`**

OpenMP 4.5 – Asynchronous Execution



The syntax of the **target** construct is as follows:

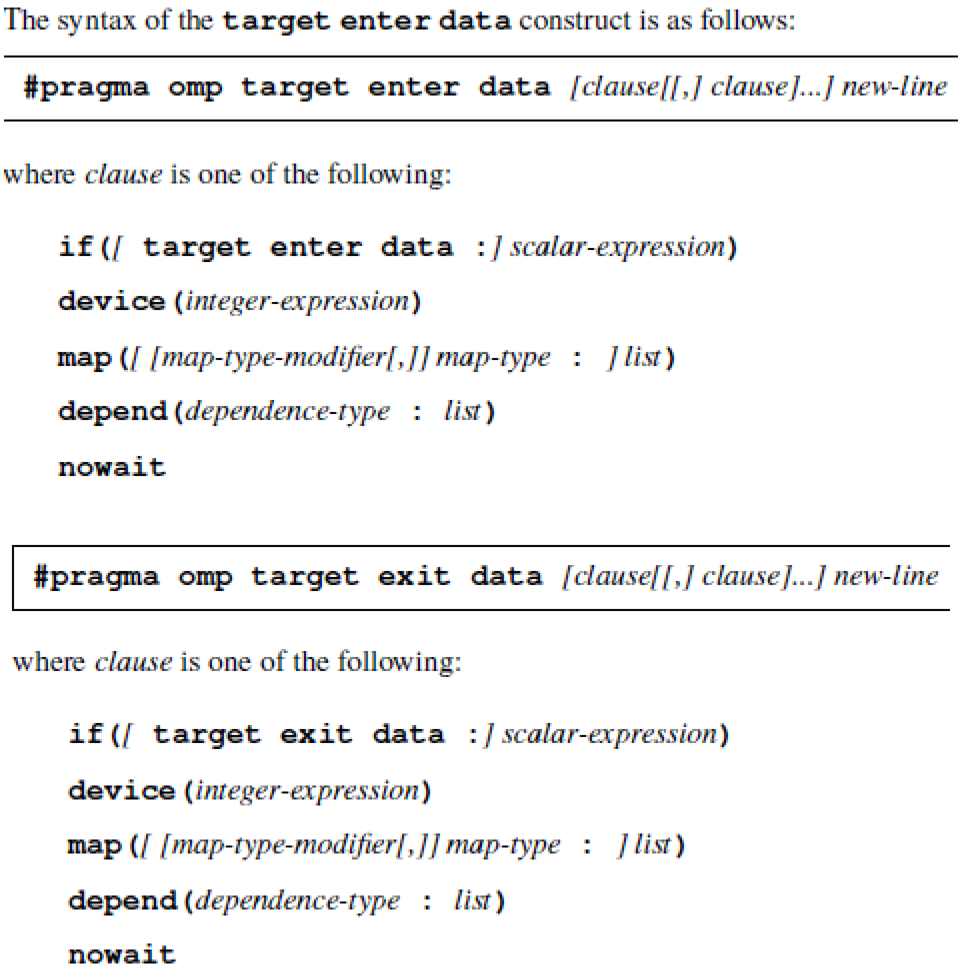
```
#pragma omp target [clause[ [,] clause] ... ] new-line
structured-block
```

where *clause* is one of the following:

```
if([ target :] scalar-expression)
device(integer-expression)
private(list)
firstprivate(list)
map([[map-type-modifier[,]] map-type: ] list)
is_device_ptr(list)
defaultmap(tofrom:scalar)
nowait
depend(dependence-type: list)
```

- The **nowait** clause indicates that the encountering thread does not wait for the target region to complete.
- A host task is generated that encloses the target region.
- The **depend** clause can be used for synchronization with other tasks

OpenMP 4.5 – unstructured data mapping



The syntax of the **target enter data** construct is as follows:

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

where *clause* is one of the following:

```
if([ target enter data :] scalar-expression)
device (integer-expression)
map ([ [map-type-modifier[,]] map-type  :  ] list)
depend (dependence-type  :  list)
nowait
```

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

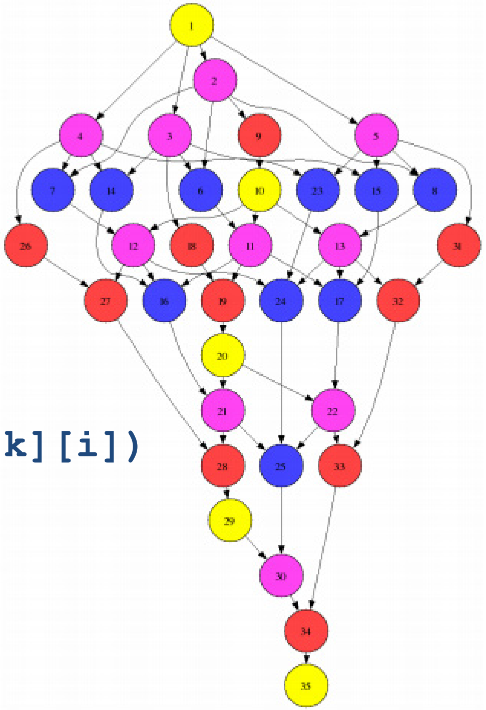
where *clause* is one of the following:

```
if([ target exit data :] scalar-expression)
device (integer-expression)
map ([ [map-type-modifier[,]] map-type  :  ] list)
depend (dependence-type  :  list)
nowait
```

- Structured **target data** construct is too restrictive and does not fit for C++ (de)constructors.
- **target enter data**
  - Map variable to a device
- **target exit data**
  - Map variable from a device

Task Dependencies

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

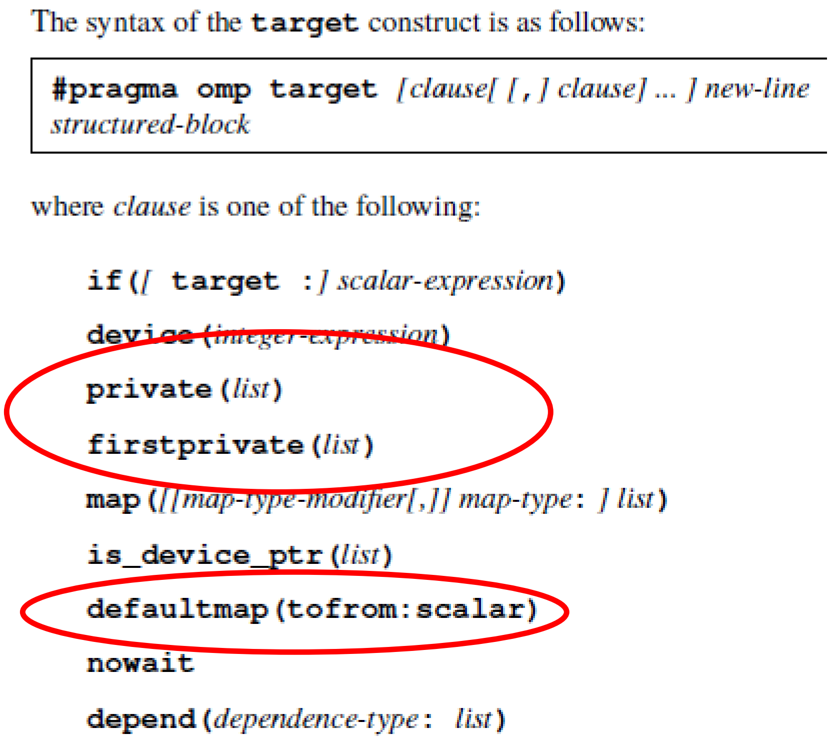


* image from BSC

Tasks and Target 4.5 Example

```
#pragma omp declare target
#include <stdlib.h>
#include <omp.h>
extern void compute(float *, float *, int);
#pragma omp end declare target

void vec_mult_async(float *p, float *v1, float *v2, int N, int dev)
{
   int i;

   #pragma omp target enter data map(alloc: v1[:N], v2[:N])

   #pragma omp target nowait depend(out: v1, v2)
   {
      compute(v1,v2,N);
   }

   #pragma omp task
   other_work(); // execute asynchronously on host device

   #pragma omp target map(from:p[0:N]) nowait depend(in: v1, v2)
   {
      #pragma omp distribute parallel for
      for (i=0; i<N; i++)
         p[i] = v1[i] * v2[i];
   }

   #pragma taskwait

   #pragma omp target exit data map(release: v1[:N], v2[:N])
}
```

Performance Oriented Changes



The syntax of the **target** construct is as follows:

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

where *clause* is one of the following:

```
if([ target :] scalar-expression)
device(integer-expression)
private(list)
firstprivate(list)
map([[map-type-modifier[,]] map-type: ] list)
is_device_ptr(list)
defaultmap(tofrom:scalar)
nowait
depend(dependence-type: list)
```

- By default scalar variables are now firstprivate
- Use **defaultmap** clause to change default behavior for scalars
- By default pointer variables in array sections are private

Device pointer

- **New clauses**
  - **#pragma omp target data … use_device_ptr(list) ..**
  - **#pragma omp target … is_device_ptr(list) ..**

- **New API**
  - **void* omp_target_alloc(size_t** *size*, **int** *device_num***);**
  - **void omp_target_free(void *** *device_ptr*, **int** *device_num***);**
  - **int omp_target_is_present(void *** *ptr*, **size_t** *offset*, **int** *device_num***);**
  - **int omp_target_memcpy(void *** *dst*, **void *** *src*, **size_t** *length*, **size_t** *dst_offset*, **size_t** *src_offset*, **int** *dst_device_num*, **int** *src_device_num***);**
  - **int omp_target_memcpy_rect(void *** *dst*, **void *** *src*, **size_t** *element_size*, **int** *num_dims*, **const size_t*** *volume*, **const size_t*** *dst_offsets*, **const size_t*** *src_offsets*, **const size_t*** *dst_dimensions*, **const size_t*** *src_dimensions*,**int** *dst_device_num*, **int** *src_device_num***);**
  - **int omp_target_associate_ptr(void *** *host_ptr*, **void *** *device_ptr*, **size_t** *size*, **size_t** *device_offset*, **int** *device_num***);**
  - **int omp_target_disassociate_ptr(void *** *ptr*, **int** *device_num***);**
  - **int omp_get_initial_device (void)**

Combined Constructs

- – **#pragma omp target parallel**
- – **#pragma omp target parallel for**
- – **#pragma omp target parallel for simd**
- – **#pragma omp target simd**

Declare target link

#pragma omp declare target *clause[[,] clause] ...] new-line*

where *clause* is one of the following:

*[***to***]* **(***extended-list***)**

link(*list***)**

The list items of a **link** clause are not mapped by the **declare target** directive. Instead, their mapping is deferred until they are mapped by **target data** or **target** constructs. They are mapped only for such regions.

```
#pragma omp declare target
int foo();
#pragma omp end declare target

extern int a;
#pragma omp declare target link(a)

int foo()
{
   a = 1;
}
main()
{
   #pragma omp target map(a)
   {
       foo();
   }
}
```

Clause changes (1)

- **If clause**
  - **if(***[directive-name-modifier* **:***]* *scalar-expression***)**

The effect of the **if** clause depends on the construct to which is it applied. For combined or composite constructs, the **if** clause only applies to the semantics of the construct named in the directive-name-modifier if one is specified. If no directive-name-modifier is specified for a combined or composite construct then the **if** clause applies to all constructs to which an if clause can apply.

Eg:-

```
 #pragma omp target data if(target data : 1)
#pragma omp task if(task : 1)
#pragma omp target parallel for if( target : 1 ) if ( parallel: 0 )
```

`if` Clause Example

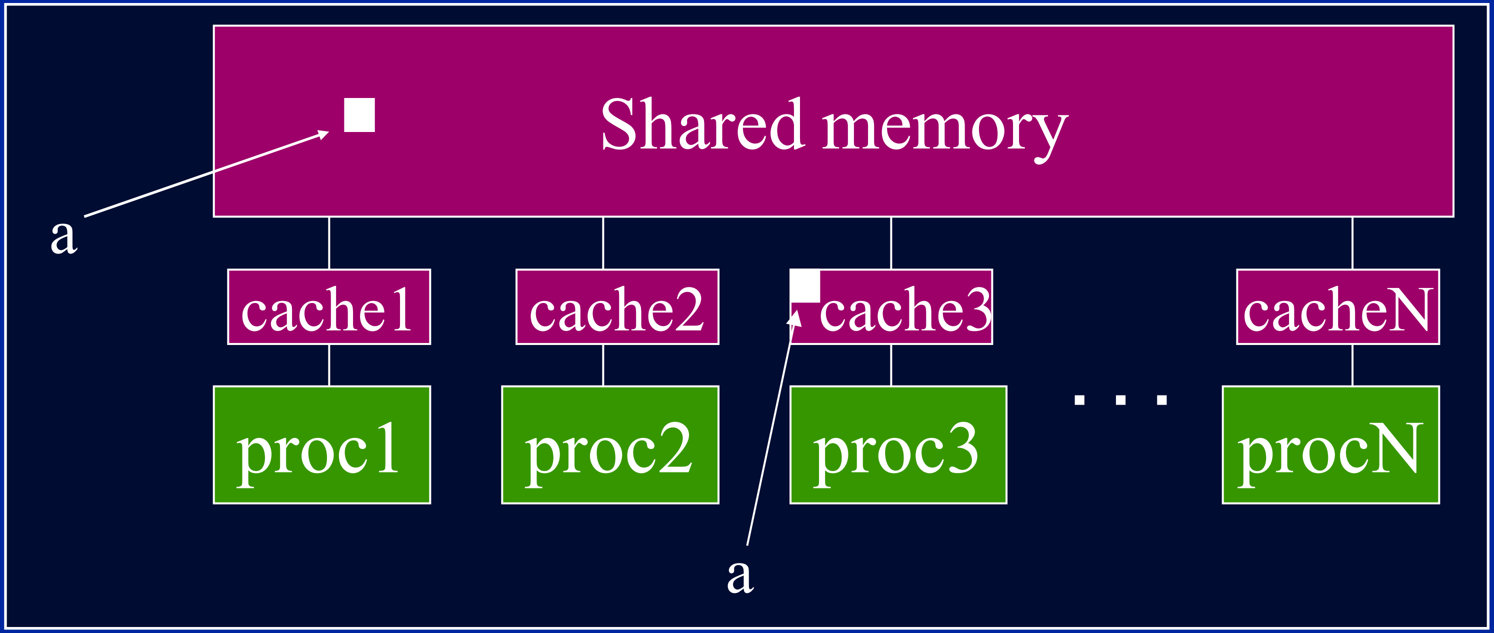
```
#define THRESHOLD1 1000000
#define THRESHOLD2 1000

extern void init(float*, float*, int);
extern void output(float*, int);

void vec_mult(float *p, float *v1, float *v2, int N)
{
   int i;
   init(v1, v2, N);

   #pragma omp target if(N>THRESHOLD1) \\
          map(to: v1[0:N], v2[:N]) map(from: p[0:N])
   #pragma omp parallel for if(N>THRESHOLD2)
   for (i=0; i<N; i++)
     p[i] = v1[i] * v2[i];
   output(p, N);
}
```

- The `if` clause on the `target` construct indicates that if the variable N is smaller than a given threshold, then the `target` region will be executed by the host device.

- The `if` clause on the `parallel` construct indicates that if the variable N is smaller than a second threshold then the parallel region is inactive.

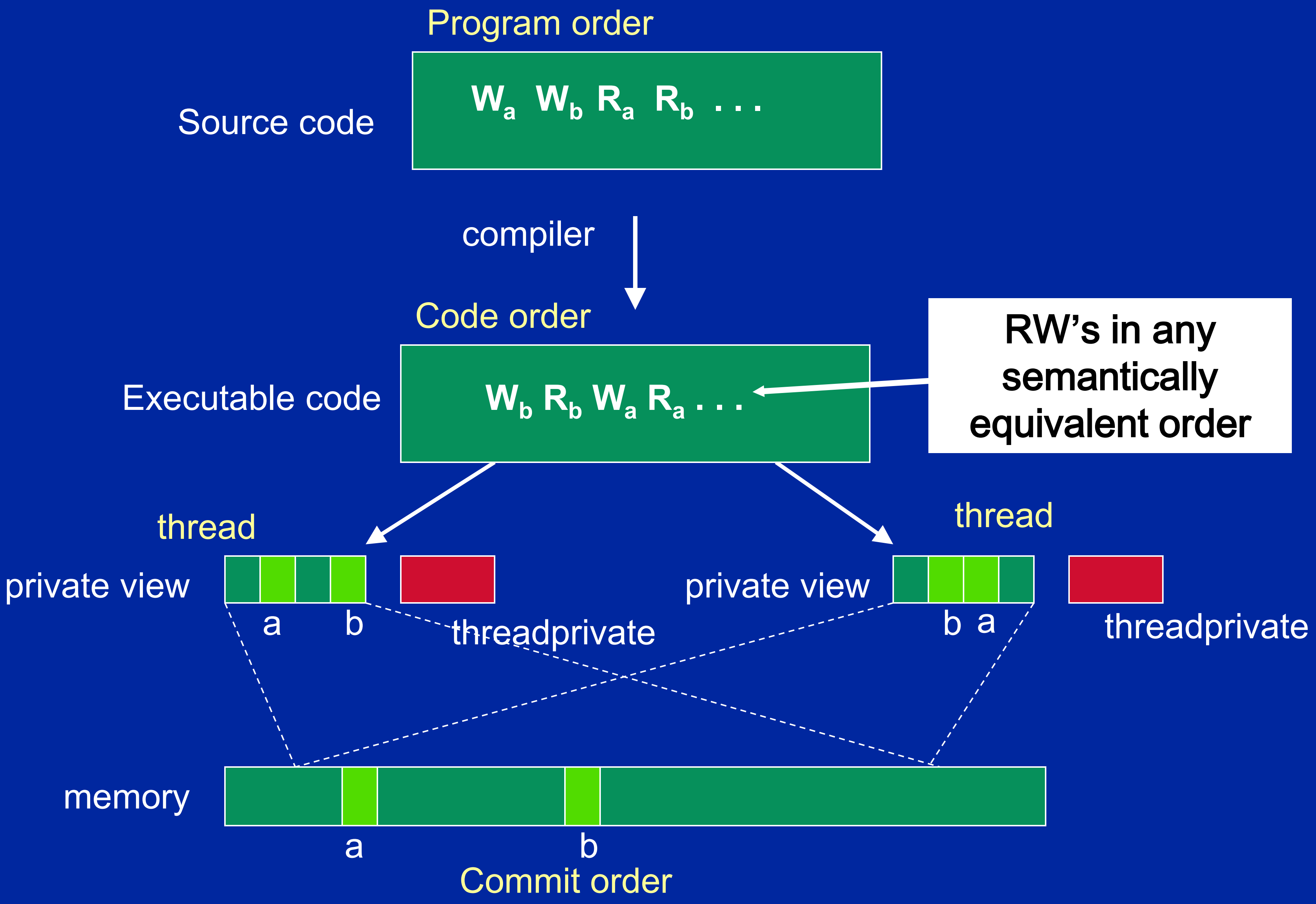
OpenMP memory model

- OpenMP supports a shared memory model.
- All threads share an address space, but it can get complicated:



- A memory model is defined in terms of:
  - Coherence: Behavior of the memory system when a single address is accessed by multiple threads.
  - Consistency: Orderings of reads, writes, or synchronizations (RWS) with various addresses and by multiple threads.

OpenMP Memory Model: Basic Terms



Program order

Source code

W_a W_b R_a R_b . . .

compiler

Code order

Executable code

W_b R_b W_a R_a . . .

RW's in any semantically equivalent order

thread

private view

a b

threadprivate

thread

private view

b a

threadprivate

memory

a b

Commit order

Consistency: Memory Access Re-ordering

- Re-ordering:
  - Compiler re-orders program order to the code order
  - Machine re-orders code order to the memory commit order
- At a given point in time, the "private view" seen by a thread may be different from the view in shared memory.
- Consistency Models define constraints on the orders of Reads (R), Writes (W) and Synchronizations (S)
  - … i.e. how do the values "seen" by a thread change as you change how ops follow (→) other ops.
  - Possibilities include:
    - R→R, W→W, R→W, R→S, S→S, W→S

Consistency

- Sequential Consistency:
  - In a multi-processor, ops (R, W, S) are sequentially consistent if:
    - They remain in program order for each processor.
    - They are seen to be in the same overall order by each of the other processors.
  - Program order = code order = commit order
- Relaxed consistency:
  - Remove some of the ordering constraints for memory ops (R, W, S).

OpenMP and Relaxed Consistency

- OpenMP defines consistency as a variant of weak consistency:
  - Can not reorder S ops with R or W ops on the same thread
    - Weak consistency guarantees

      S→W,   S→R , R→S, W→S, S→S
- The Synchronization operation relevant to this discussion is flush.

Flush

- Defines a sequence point at which a thread is guaranteed to see a consistent view of memory with respect to the “flush set”.
- The flush set is:
  - “all thread visible variables” for a flush construct without an argument list.
  - a list of variables when the “flush(list)” construct is used.
- The action of Flush is to guarantee that:
  - All R,W ops that overlap the flush set and occur prior to the flush complete before the flush executes
  - All R,W ops that overlap the flush set and occur after the flush don’t execute until after the flush.
  - Flushes with overlapping flush sets can not be reordered.

Memory ops: R = Read, W = write, S = synchronization

Synchronization: flush example

- Flush forces data to be updated in memory so other threads see the most recent value

```
double A;

A = compute();

#pragma omp flush(A);   // flush to memory to make sure
other          //  threads can pick up the right value
```

Note: OpenMP’s flush is analogous to a fence in other shared memory API’s.

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
  - whenever a lock is set or unset

  ….

  (but not at entry to worksharing regions or entry/exit of master regions)

What is the Big Deal with Flush?

- Compilers routinely reorder instructions implementing a program
  - This helps better exploit the functional units, keep machine busy, hide memory latencies, etc.
- Compiler generally cannot move instructions:
  - past a barrier
  - past a flush on all variables
- But it can move them past a flush with a list of variables so long as those variables are not accessed
- Keeping track of consistency when flushes are used can be confusing … especially if “flush(list)” is used.

Note: the flush operation does not actually synchronize different threads. It just ensures that a thread’s values are made consistent with main memory.

Pair wise synchronizaion in OpenMP

- OpenMP lacks synchronization constructs that work between pairs of threads.
- When this is needed you have to build it yourself.
- Pair wise synchronization
  - Use a shared flag variable
  - Reader spins waiting for the new flag value
  - Use flushes to force updates to and from memory

Example: producer consumer

```
int main()
{
    double *A, sum, runtime;     int numthreads, flag = 0;
    A = (double *)malloc(N*sizeof(double));
    #pragma omp parallel sections
    {
      #pragma omp section
       {
          fill_rand(N, A);
          #pragma omp flush
          flag = 1;
          #pragma omp flush (flag)
       }
       #pragma omp section
       {
          #pragma omp flush (flag)
          while (flag == 0){
               #pragma omp flush (flag)
          }
          #pragma omp flush
          sum = Sum_array(N, A);
       }
     }
}
```

Use flag to Signal when the "produced" value is ready

Flush forces refresh to memory. Guarantees that the other thread sees the new value of A

Flush needed on both "reader" and "writer" sides of the communication

Notice you must put the flush inside the while loop to make sure the updated flag variable is seen

The problem is this program technically has a race … on the store and later load of flag.

Atomics and synchronization flags

```
int main()
{   double *A, sum, runtime;
    int numthreads, flag = 0, flg_tmp;
    A = (double *)malloc(N*sizeof(double));
    #pragma omp parallel sections
    {
      #pragma omp section
       {  fill_rand(N, A);
          #pragma omp flush
          #pragma atomic write
                 flag = 1;
          #pragma omp flush (flag)
       }
       #pragma omp section
       {  while (1){
             #pragma omp flush(flag)
             #pragma omp atomic read
                  flg_tmp= flag;
              if (flg_tmp==1) break;
          }
          #pragma omp flush
          sum = Sum_array(N, A);
       }
     }
}
```

This program is truly race free … the reads and writes of flag are protected so the two threads can not conflict.

Data sharing: Threadprivate

- Makes global data private to a thread
  - Fortran: COMMON blocks
  - C: File scope and static variables, static class members
- Different from making them PRIVATE
  - with PRIVATE global variables are masked.
  - THREADPRIVATE preserves global scope within each thread
- Threadprivate variables can be initialized using COPYIN or at time of definition (using language-defined initialization capabilities).

A threadprivate example (C)

Use threadprivate to create a counter for each thread.

```
int counter = 0;
#pragma omp threadprivate(counter)

int increment_counter()
{
    counter++;
    return (counter);
}
```

Data Copying: Copyin

You initialize threadprivate data using a copyin clause.

```
      parameter (N=1000)
      common/buf/A(N)
!$OMP THREADPRIVATE(/buf/)

C Initialize the A array
      call init_data(N,A)

!$OMP PARALLEL COPYIN(A)

 … Now each thread sees threadprivate array A initialied
 … to the global value set in the subroutine init_data()

!$OMP END PARALLEL

end
```