CSC367 Parallel computing

Lecture 10: Shared Memory Programming models: OpenMP

Thanks to Scinet, NSF XSEDE, James Demmel, and Aydin Buluc

Next up ...

- Shared memory architecture
- Parallel programing models: shared memory
- Pthreads: Synchronization, Races, Locks
- OpenMP
- Cache coherency

Summary of Programming with Threads

- POSIX Threads are based on OS features
- Pitfalls
  - Overhead of thread creation is high (1-loop iteration probably too much)
  - Data race bugs are very nasty to find because they can be intermittent
  - Deadlocks are usually easier, but can also be intermittent
- OpenMP is commonly used today as an alternative
  - Helps with some of these, but doesn’t make them disappear

What is OpenMP?

- OpenMP = Open specification for Multi-Processing
  - openmp.org – Talks, examples, forums, etc.
  - Spec controlled by the ARB
- Motivation: capture common usage and simplify programming
- OpenMP Architecture Review Board (ARB)
  - A nonprofit organization that controls the OpenMP Spec
  - Latest spec: OpenMP 4.5 (Nov. 2015), working on 5.0
- High-level API for programming in C/C++ and Fortran
  - Preprocessor (compiler) directives (~ 80%)

    #pragma omp *construct [clause [clause …]]*
  - Library Calls (~ 19%)

    #include <omp.h>
  - Environment Variables (~ 1%)

    all caps!

A Programmer's View of OpenMP

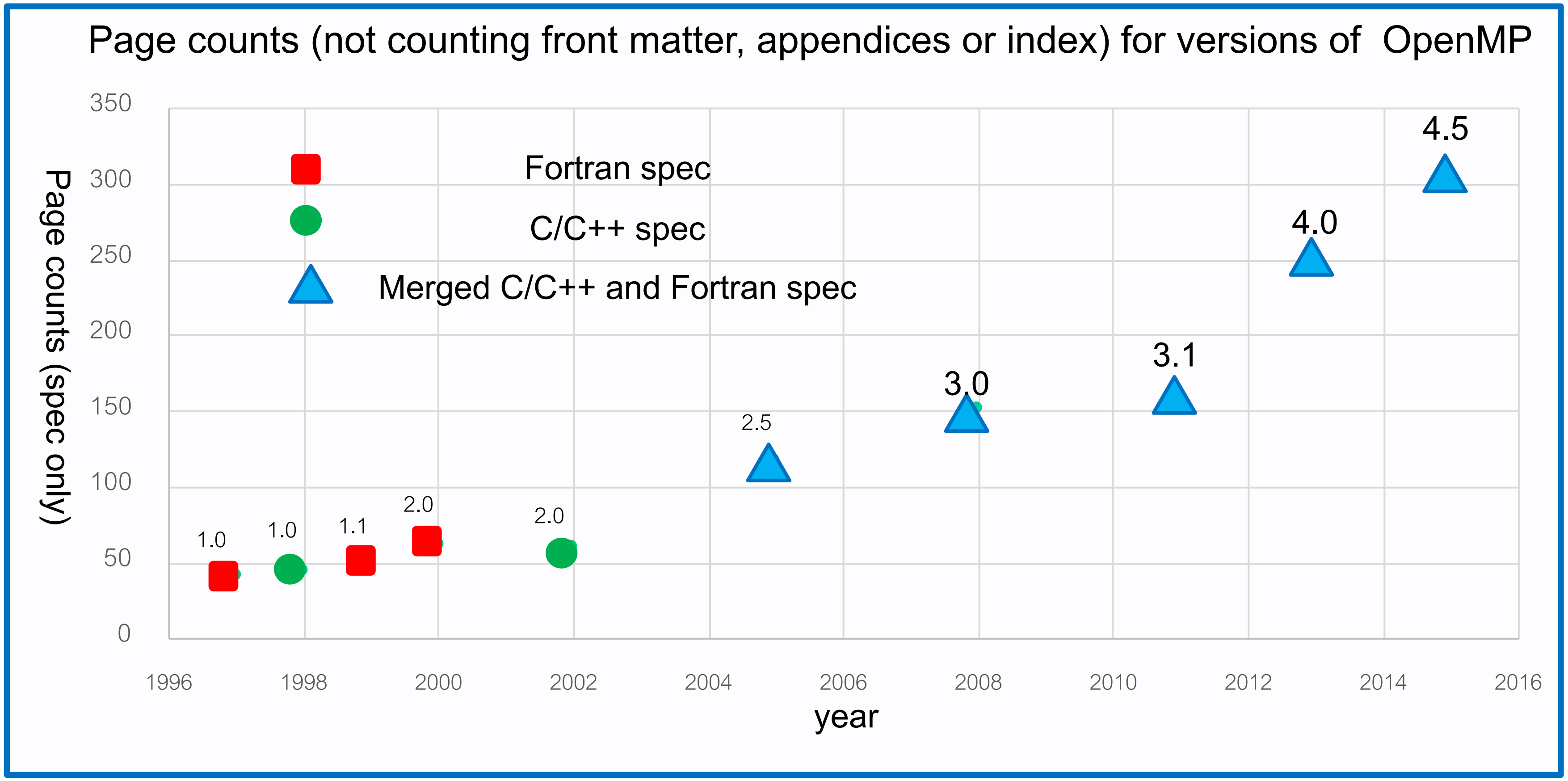
- OpenMP is a portable, threaded, shared-memory programming *specification* with "light" syntax
  - Requires compiler support (C, C++ or Fortran)

- OpenMP will:
  - Allow a programmer to separate a program into *serial regions* and *parallel regions,* rather than P concurrently-executing threads.
  - Hide stack management
  - Provide synchronization constructs

- OpenMP will not:
  - Parallelize automatically
  - Guarantee speedup
  - Provide freedom from data races

The growth of complexity in OpenMP

- OpenMP started out in 1997 as a simple interface for the application programmers more versed in their area of science than computer science. The complexity has grown considerably over the years!



The complexity of the full spec is overwhelming, so we focus on the 16 constructs most OpenMP programmers restrict themselves to … the so called "OpenMP Common Core"

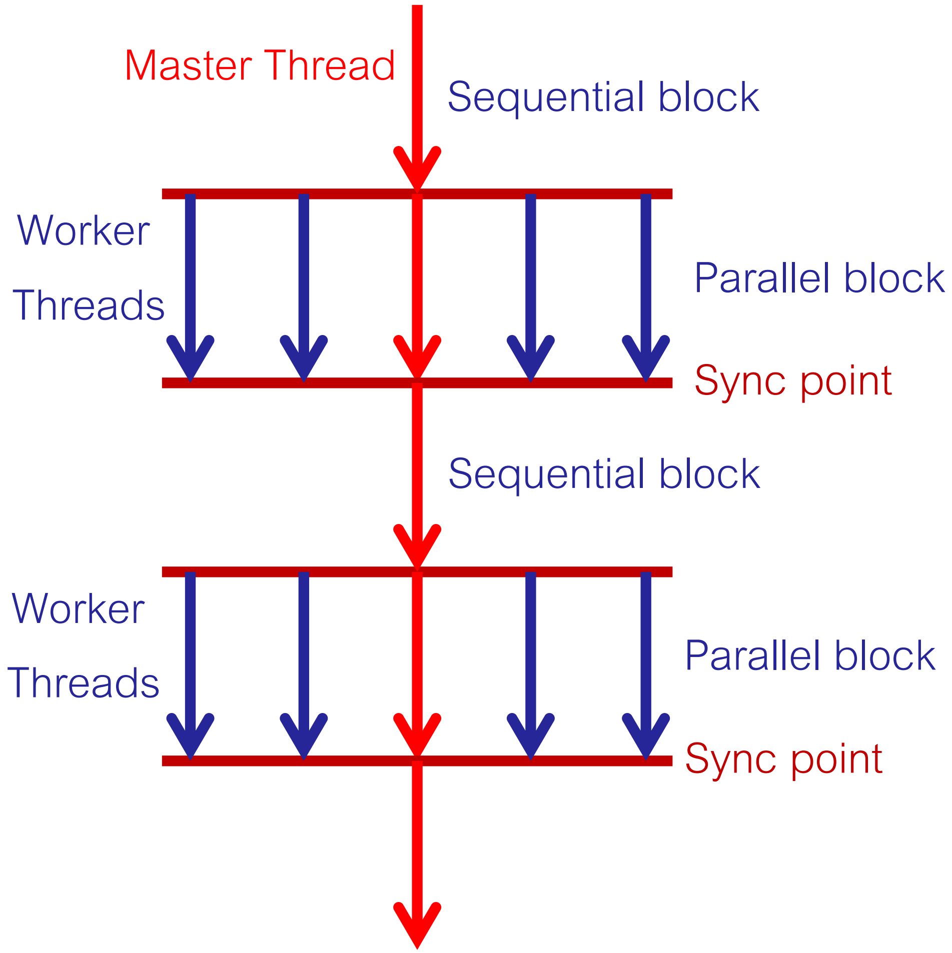
The OpenMP Common Core: Most OpenMP programs only use these 19 items

OpenMP pragma, function, or clause	Concepts
#pragma omp parallel	Parallel region, teams of threads, structured block, interleaved execution across threads
int omp_get_thread_num() int omp_get_num_threads()	Create threads with a parallel region and split up the work using the number of threads and thread ID
double omp_get_wtime()	Speedup and Amdahl's law.
setenv OMP_NUM_THREADS N	Internal control variables. Setting the default number of threads with an environment variable
#pragma omp barrier #pragma omp critical	Synchronization and race conditions.
#pragma omp for #pragma omp parallel for	Worksharing, parallel loops, loop carried dependencies
reduction(op:list)	Reductions of values across a team of threads
schedule(dynamic [,chunk]) schedule (static [,chunk])	Loop schedules, loop overheads and load balance
private(list), firstprivate(list), shared(list)	Data environment
nowait	Disabling implied barriers on workshare constructs
#pragma omp single	Workshare with a single thread
#pragma omp task and #pragma omp section #pragma omp taskwait	Tasks including the data environment for tasks.

The course project might use some other routines, see the following link for more routines
https://computing.llnl.gov/tutorials/openMP/

Execution model: Fork-and-Join

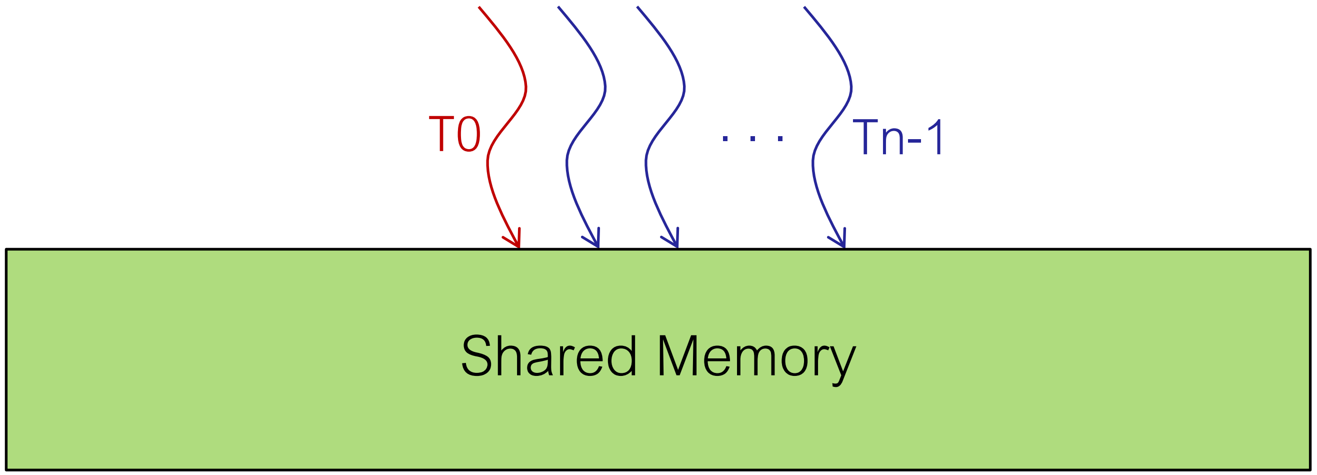
- Start with a single (master) thread
- Fork: Create a group of worker threads
  - Everything in a parallel block is executed by every thread
- Join: All threads synchronize at the end of the parallel block
  - Execution continues with only the initial (master) thread



- Threads can do the exact same work, share the same tasks (work sharing), or perform distinct tasks in parallel!

Reminder: Shared Memory Model

- All worker threads share the same address space



- OpenMP provides the ability to declare variables private or shared within a parallel block (more on this later...)

OpenMP Programming Basics

- Programming model: provide "hints" or "directives" to the compiler as to what you intend to parallelize
  - C/C++ #**pragma** compiler directives, followed by various clauses:

```
#pragma omp directivename [clause list]
```

- Must include OpenMP function headers

```
#include <omp.h>
```

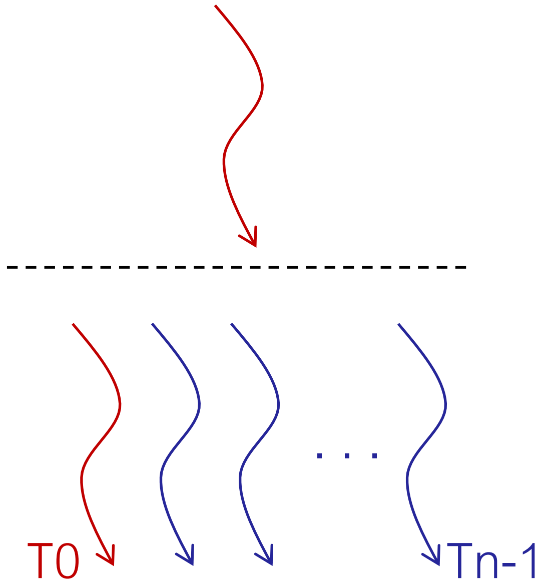
- Compile like a regular C program, link with omp library

```
gcc -fopenmp ... [-std=c99/gnu99]
```

- Debug with gdb and valgrind, or dedicated parallel debuggers e.g., TotalView, DDT, etc. DDT should be on Scinet.

Parallel directive

- Regular serial execution until it hits a `parallel` directive
    - Creates a group of threads, main thread becomes the *master* of the group



```
// Serial code

#pragma omp parallel [clause list]
{
 // Code executed by all threads

}
```

A first parallel program

- Default number of threads to be spawned is stored in the environment variable OMP_NUM_THREADS
  - Useful to set a default

```
int main(int argc, char *argv[]) {
   printf("Starting a parallel region, spawning threads\n");

   #pragma omp parallel
   {
      printf("Hello world\n");
   }

   return 0;
}


$ export OMP_NUM_THREADS=8
$ gcc -o hello hello.c -fopenmp
```

- All threads execute the exact same code from the parallel region

A first parallel program

- Parallel region spawns a block of code or a line (no braces needed for latter)

```
int main() {
   printf("Starting a parallel region, spawning threads\n");

   #pragma omp parallel
   {
      printf("Hello world\n");
   }
   printf("Bye world\n");

   return 0;
}


$ export OMP_NUM_THREADS=8
$ gcc -o hello hello.c -fopenmp
```

If you do not set the number of threads, X number of threads will be automatically spawned in the parallel region, where X is the number of cores (or the number of threads with hyperthreading if enabled)

OpenMP: language extension + library

- Language extensions: `#pragma omp` (ignored if not compiled with `-fopenmp`), great feature for monitoring the behavior of the serial version of your code

- Library functions (must include `omp.h`, even if compiled with `-fopenmp`):

```
int omp_get_num_threads(); /* # of threads running when this is invoked */
                           /* refers to closest enclosing parallel block*/

void omp_set_num_threads(int n); /* # of threads to use in the next
                                    parallel section */

int omp_get_max_threads(); /* max number of threads that can be created */

int omp_get_thread_num(); /* thread id in a group of threads */

int omp_get_num_procs(); /* number of processors available */

int omp_in_parallel(); /* non-zero if called within a parallel region */
```

A first parallel program

- Threads run in parallel - no guarantees about ordering

```
int main() {
  printf("Starting a parallel region, spawning threads\n");

  #pragma omp parallel
  {
     printf("Hello world, I am thread %d out of %d running threads!\n",
            omp_get_thread_num(),
            omp_get_num_threads());
  }
  printf("There are %d threads running!\n", omp_get_num_threads());
                                                  1
  return 0;
}
```

- Notice anything? The last printf is only visited by the main thread and prints `There are 1 threads running!`

Parallel directive clauses

- Conditional Parallelization: `if` clause (only one!)
    - Only create threads if an expression holds

```
#pragma omp parallel if(to_parallelize == 1)
```

- Degree of concurrency: `num_threads` clause
    - How many threads to spawn, overrides the default OMP_NUM_THREADS

```
#pragma omp parallel num_threads(8)
```

- Data handling: `private/firstprivate/shared` clauses (different visibility)

```
#pragma omp parallel private(v1) shared(v2,v3) firstprivate(v4)
```

Up Next…

Variable semantics: shared, private, firstprivate

The reduction directive

The Omp for directive

Examples on the above

Variable semantics - summary

- Semantics of each variable
  - Shared: all threads share the same copy of the variable(s) reference
  - Private: variable(s) are local to each thread implies a copy!
  - Firstprivate: like private, but value of each copy is initialized to the value before the parallel directive

```
#pragma omp parallel private(v1) shared(v2,v3) firstprivate(v4)
```

Data sharing

- 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 private 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 private 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

Careful with variable state

- Think carefully about the intended variable visibility between threads
- Should a variable be shared or private?
  - Private: a thread can modify its own copy without impacting other threads
  - Shared: all threads see the exact same copy, tricky if the data is not read-only
- Dramatic impact on correctness if you get this wrong!

Default state

- Can specify default state using `default(...)` clause
  - `default(shared)` = by default, a variable is shared by all threads
  - `default(none)` = must explicitly specify how every single variable used in parallel region should be handled, otherwise compile errors raised
- If not explicitly indicated
  - Variables declared *within* a parallel block are implicitly private
  - Variables declared *outside* a parallel block become shared when parallel region starts (with some exceptions like *some* loop counters .. more later)
- Safer to use default(none) to force you to specify intended visibility
  - Recommended to avoid possible bugs from unintentionally sharing data

Reduction

- Important primitive – supported seamlessly using `reduction` clause
  - Multiple local copies of a variable are combined into a single copy at the master when the parallel block ends

    `reduction(operator: variable list)`

  - Operators: +, *, -, &, |, ^, &&, ||

atomic reduce (synchronized)

```
int s = 0;
#pragma omp parallel reduction(+:s) num_threads(8)
{
    // compute sum s
    s += ...
}
// variable s now has the sum of all local dim copies
```

Reduction: Simple example

- Calculate the sum of the thread ids

```
int sum = 0;
#pragma omp parallel reduction(+: sum) num_threads(8)
{
   int tid = omp_get_thread_num();

   sum += tid;
}
printf("Sum of thread ids = %d\n", sum);
```

Each thread gets a private sum copy

- Simple but useful construct for more complex computations (as we'll see later ..)

Reduction: More complex example

- Calculate the dot product of two arrays 'a' and 'b' of length 'size'
- Can use the reduction clause for the parallel directive

```
int dotprod = 0;
#pragma omp parallel shared(size, a, b) \
        reduction(+: dotprod) num_threads(8)
{
   int nthr = omp_get_num_threads();
   int chunk = (size+nthr-1) / nthr;
   int tid = omp_get_thread_num();

   for(int i = tid*chunk;
           i < (tid+1)*chunk && i < size; i++) {
      dotprod += a[i] * b[i];
   }
}
```

Each thread gets a private dotprod copy

Loop counter i is implicitly private

- Notice that we have to explicitly partition the data, and make sure that each thread only operates on its assigned chunk

For loops in OpenMP

- Previous example: Every thread executes the same loop, just on different value ranges of a and b
- A simple example:

```c
int i, size=8;
int *a = (int*)malloc(size*sizeof(int));
for(i = 0; i < size; i++) {
   a[i] = i+1;
}

#pragma omp parallel shared(size, a) private(i) num_threads(8)
{
   int tid = omp_get_thread_num();
   for(i = 0; i < size; i++) {
       printf("TID[%d] - a[%d] = %d\n", tid, i, a[i]);
   }
}
```

- Kind of useless work because each thread executes all iterations of the loop.
- What is the output?

For loops in OpenMP

- Previous example: Every thread executes the same loop, just on different value ranges of a and b
- A simple example:

```
int i, size=8;
int *a = (int*)malloc(size*sizeof(int));
for(i = 0; i < size; i++) {
   a[i] = i+1;
}

#pragma omp parallel shared(size, a) private(i) num_threads(8)
{
   int tid = omp_get_thread_num();
   for(i = 0; i < size; i++) {
       printf("TID[%d] - a[%d] = %d\n", tid,
   }
}
```

```
TID[1] - a[0] = 1
TID[1] - a[1] = 2
TID[1] - a[2] = 3
TID[0] - a[0] = 1
TID[0] - a[1] = 2
TID[0] - a[2] = 3
TID[0] - a[3] = 4
TID[0] - a[4] = 5
TID[0] - a[5] = 6
TID[0] - a[6] = 7
      ….
```

- Kind of useless work.

size * num_threads iterations
each thread loop over entire for loop

Concurrent tasks – loop scheduling

- Use `parallel` directive to create concurrency across iterations
  - Recall task parallelism!
- Then, automatically divvy up the iterations across threads using `for` directive

```
#pragma omp for [clause list]
// for loop
```

- Clauses:
  - `private, firstprivate, lastprivate`
  - `reduction`
  - `schedule`
  - `nowait`
  - `ordered`

  - private and firstprivate – same semantics as for parallel directive
  - `lastprivate` handles writing back a single copy from multiple copies

Example: for directive

- A simple example:

```
int i, size=8;
int *a = (int*)malloc(size*sizeof(int));
for(i = 0; i < size; i++) {
   a[i] = i+1;
}

#pragma omp parallel shared(size, a) private(i) num_threads(8)
{
   #pragma omp for
   int tid = omp_get_thread_num();
   for(i = 0; i < size; i++) {
       printf("TID[%d] - a[%d] = %d\n", tid, i, a[i]);
   }
}
```

Compiler error

- What will this print?

Example: for directive

- A simple example:

```
int i, size=8;
int *a = (int*)malloc(size*sizeof(int));
for(i = 0; i < size; i++) {
   a[i] = i+1;
}

#pragma omp parallel shared(size, a) private(i) num_threads(8)
{
   int tid = omp_get_thread_num();
   #pragma omp for
   for(i = 0; i < size; i++) {
       printf("TID[%d] - a[%d] = %d\n", tid, i, a[i]);
   }
}
```

- What will this print?

Example: for directive

- A simple example:

```
int i, size=8;
int *a = (int*)malloc(size*sizeof(int));
for(i = 0; i < size; i++) {
   a[i] = i+1;
}

#pragma omp parallel shared(size, a) private(i) num_threads(8)
{
   int tid = omp_get_thread_num();
   #pragma omp for
   for(i = 0; i < size; i++) {
       printf("TID[%d] - a[%d] = %d\n", tid, i, a[i]);
   }
}
```

`size` iterations in total for all threads

- What will this print?
  - Loop iterations partitioned across threads now
  - Each thread takes care of different iterations

```
TID[1] - a[1] = 2
TID[7] - a[7] = 8
TID[4] - a[4] = 5
TID[6] - a[6] = 7
TID[3] - a[3] = 4
TID[5] - a[5] = 6
TID[0] - a[0] = 1
TID[2] - a[2] = 3
```

How omp for makes loop parallelism easy

An example to how how parallelizing a loop can be made easy with **omp for**

(A) Sequential code

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

(B) 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];}
}
```

(C) OpenMP parallel region and a worksharing for construct replaces (B)

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

Basic approach for 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

remove dependencies !

```
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

Remove loop carried dependence

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

More examples: 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

More examples: Firstprivate clause

- Variables initialized from a shared variable

```
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

More examples: default clause

- You can put the default clause on parallel and parallel + workshare constructs.

x: has to be shared by reduction

```
#include <omp.h>
int main()
{
     int i, j=5;      double x=1.0, y=42.0;
     #pragma omp parallel for default(none) reduction(*:x)
     for (i=0;i<10;i++){
         for(j=0; j<3; j++)
               x+= foobar(i, j, y);
     }
     printf(" x is %f\n",(float)x);
}
```

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

Loop counter i is implicitly private by the #pragma parallel for

The reduction also implicitly defines the state of x

Up Next…

The Schedule clause

Multiple for directives

The omp section clause

Synchronization constructs

Scheduling work to threads

- The `schedule` clause – ways to assign iterations to threads

  `schedule(scheduling_class[, parameter])`

- Scheduling classes
  - `static`
  - `dynamic`
  - `guided`
  - `runtime`

- Recall models studied in the early lectures!

Scheduling classes

- Example: matrix multiplication (assume n x n square)

```
for(i = 0; i < n; i++) {
    for(j = 0; j < n; j++) {
        c[i][j] = 0;
        for(k = 0; k < n; k++) {
            c[i][j] += a[i][k] * b[k][j];
        }
    }
}
```

Static scheduling

- Split iterations into equal chunks of size S, assign to threads round-robin
    - If no chunk size specified, divide into as many chunks as threads

```
#pragma omp parallel default(none)
        shared(a, b, c, n)
        num_threads(8)
{
  #pragma omp for schedule(static)
  for(int i = 0; i < n; i++) {
     for(int j = 0; j < n; j++) {
       c[i][j] = 0;
       for(int k = 0; k < n; k++) {
          c[i][j] += a[i][k] * b[k][j];
       }
     }
  }
}
```

- Outer loop is split into 8 chunks
    - e.g., for n = 1024, chunk = 128 rows => same as `schedule(static, 128)`

Dynamic scheduling

- Load per iteration may not be balanced => equally partitioned tasks may take different execution time
- Split iterations into chunks, assign new chunk to thread only when idle
  - If no chunksize specified, default chunk is single iteration

```
#pragma omp parallel default(none)
        shared(a, b, c, n)
        num_threads(8)
{
  #pragma omp for schedule(dynamic)
  for(int i = 0; i < n; i++) {
     for(int j = 0; j < n; j++) {
       c[i][j] = 0;
       for(int k = 0; k < n; k++) {
          c[i][j] += a[i][k] * b[k][j];
       }
     }
  }
}
```

Guided scheduling

- Imagine 1000 iterations and chunk size = 50 => 20 chunks
  - 16 threads => 12 threads get 1 chunk each, 4 threads get 2 each
  - Load imbalance: 12 threads are idle for possibly a long time!
- Guided scheduling idea: start with big chunks but reduce size as computation progresses
  - Chunks get smaller and smaller as computation progresses, if load gets imbalanced

    `schedule(guided[, chunksize])`
  - Parameter `chunksize` specifies the minimum size chunk to use
  - If not specified, default `chunksize` is 1

Runtime scheduling

- Delay scheduling decision until runtime
  - Environment variable OMP_SCHEDULE determines class and chunksize

- When no scheduling class is specified => implementation dependent

Restrictions on `for` directive

```
#pragma omp for
for (int i=0; i < n; i++) {
```

Must be integer type!

Must be an integer assignment

Must be a <, >, <=, or >= expression

Must have integer increments only

```
    // statements
```

No `break` instructions!

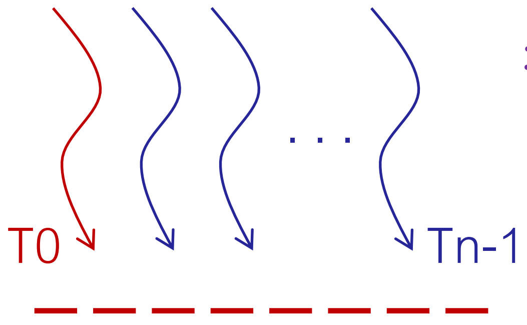
```
}
```

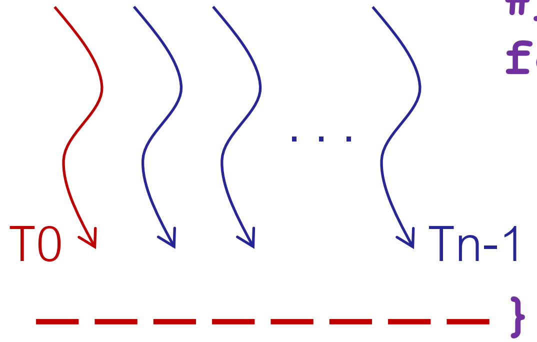
- Compiler will typically prompt you to the problem
- Refer to the OpenMP manual when it doubt!

Sequences of `for` directives

- A sequence of `for` directives: implicitly adds barrier after each one

```
#pragma omp parallel {
    // Spawn threads

     #pragma omp for
     for(...) {

       // Parallel execution of loop

     }
     #pragma omp for
     for(...) {

       // Parallel execution of loop

     }
```

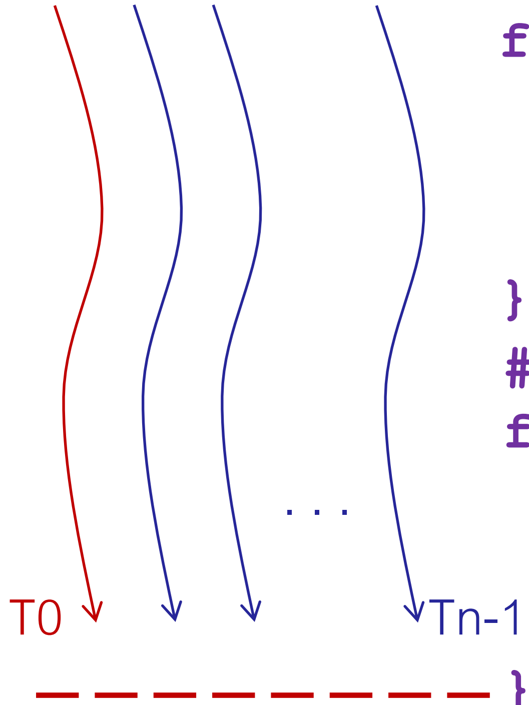




`nowait` clause

- Let threads proceed without an implicit barrier
    - Might be useful if there is no need to wait for results from previous loop

```
#pragma omp parallel
{       // Spawn threads

    #pragma omp for nowait
    for(...) {

      // Parallel execution of loop

    }
    #pragma omp for
    for(...) {

      // Parallel execution of loop

    }
}
```



Sections

- So far, threads do the same work: same block, or the same loop code
- What about diverging tasks?

```
#pragma omp parallel {
    #pragma omp sections [clauses] {
        // Specify different tasks as "section" blocks
        #pragma omp section {
            // task 1
        }
        #pragma omp section {
            // task 2
        }
        ...
        #pragma omp section {
            // task n
        }
    }
}
```

1 thread holds 1 task, specified in 1 section

Tasks scheduled in parallel

- 1 or more threads assigned to each section

- each thread gets 1 or more sections

Sections

- So far, threads do the same work: same block, or the same loop code
- What about diverging tasks?

```
#pragma omp parallel
{
    #pragma omp sections [clauses]
    {
        // Specify different tasks as "section" blocks
        #pragma omp section
        { // task 1
        }
        #pragma omp section
        { // task 2
        }
        ...
        #pragma omp section
        { // task n
        }
     }
}
```

Clauses:

- private, firstprivate, lastprivate
- reduction
- nowait

The omp task clause

`#pragma omp` task/s is another directive that is very similar to the section directive.

An implicit barrier exists at the end of the omp sections clause.

However, omp tasks does not have this implicit barrier, hence the existence of the taskwait clause. See the omp manual for more!

Merging directives

- `parallel` needed to spawn threads => merge it with `for` or `sections`

```
#pragma omp parallel shared(n)
{
  #pragma omp for
  for(int i = 0; i < n; i++) {
     // parallel execution
  }
}
```

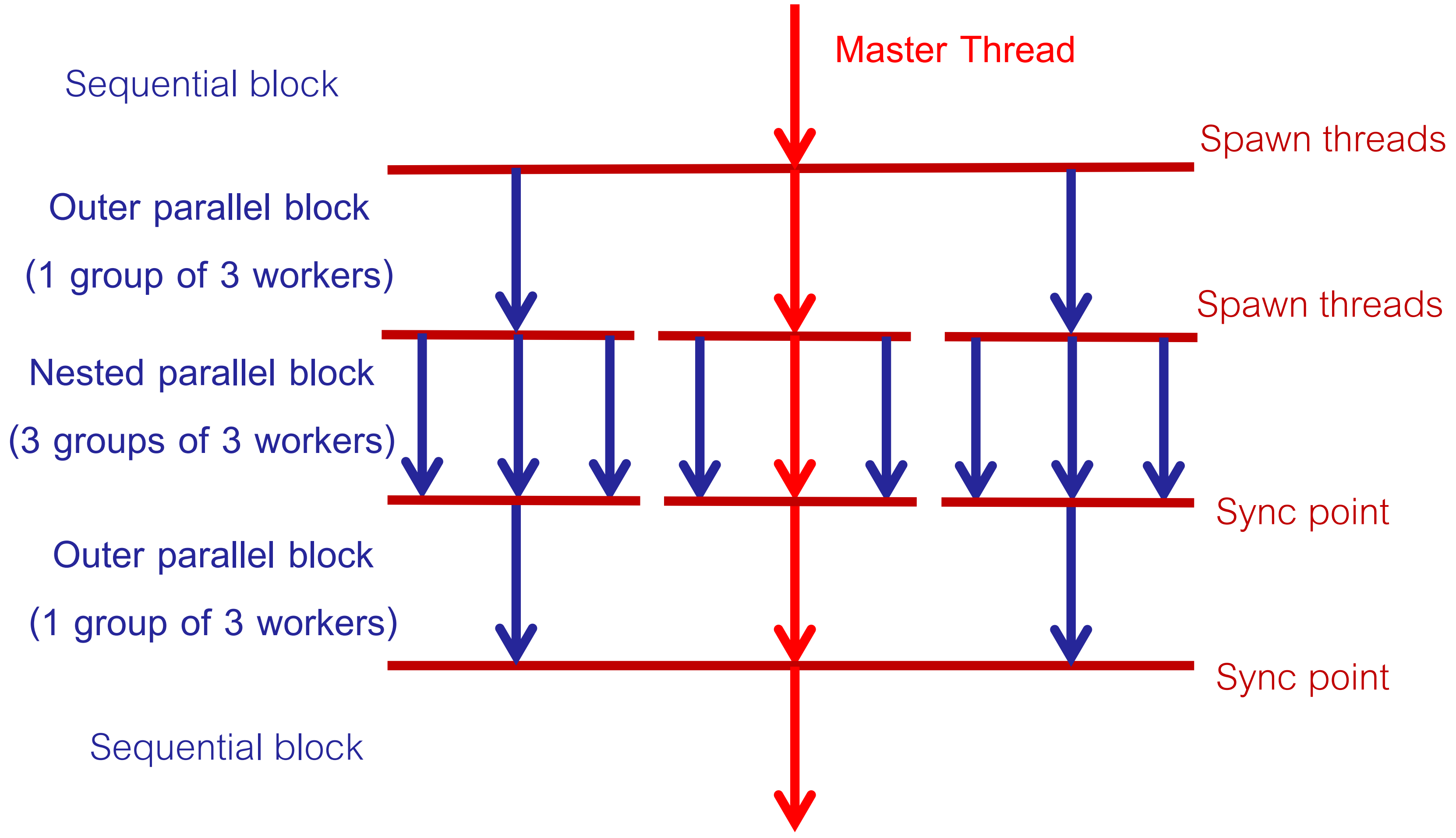
```
#pragma omp parallel for shared(n)
{
  for(int i = 0; i < n; i++) {
     // parallel execution
  }
}
```

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    { // task 1
    }
    #pragma omp section
    { // task 2
    } ...
    #pragma omp section
    {  // task n
    }
  }
}
```

```
#pragma omp parallel sections
{
   #pragma omp section
   { // task 1
   }
   #pragma omp section
   {   // task 2
   } ...
   #pragma omp section
   {   // task n
   }
}
```

Nested parallelism

- OpenMP supports arbitrarily deep nesting of parallel regions


Master Thread
Sequential block
Spawn threads
Outer parallel block
(1 group of 3 workers)
Spawn threads
Nested parallel block
(3 groups of 3 workers)
Sync point
Outer parallel block
(1 group of 3 workers)
Sync point
Sequential block