OpenMP (cont...)

Acknowledgements: OpenMP Tutorial (LLNL, XSEDE) and OpenMP by Example (Texas A&M)

OpenMP → [Directives]

Sentinel + Directive + [clause[[,] clause]...]

#pragma omp parallel private(a,b)

Directive: Parallel

#pragma omp parallel

Spawns new threads…

How do you set the max number of threads?

omp_set_num_threads()
OMP_NUM_THREADS

```
#include <omp.h>
#include <stdio.h>
int main()
{
   #pragma omp parallel
   printf("Hello from thread %d,
          nthreads %d\n",
          omp_get_thread_num(),
          omp_get_num_threads());
}
```

Parallel Clauses

Variable Scoping

1. shared (list)
2. private (list)
3. firstprivate (list)

Parallel Clauses

4. copyin (list)
5. if (*scalar expression*)
6. reduction (operator:list)

You want to dynamically switch between serial and parallel.

How would you do it?

```
#pragma omp parallel if (n>100000)
```

Other Directives

Loops

a.k.a. Parallelize a loop

```
#pragma omp parallel
  #pragma omp for … →
#pragma omp parallel for …
```

1. Limited to the loop that immediately follows it
2. DO NOT change the iteration variable

Clauses:

✓ private(list)

✓ firstprivate(list)

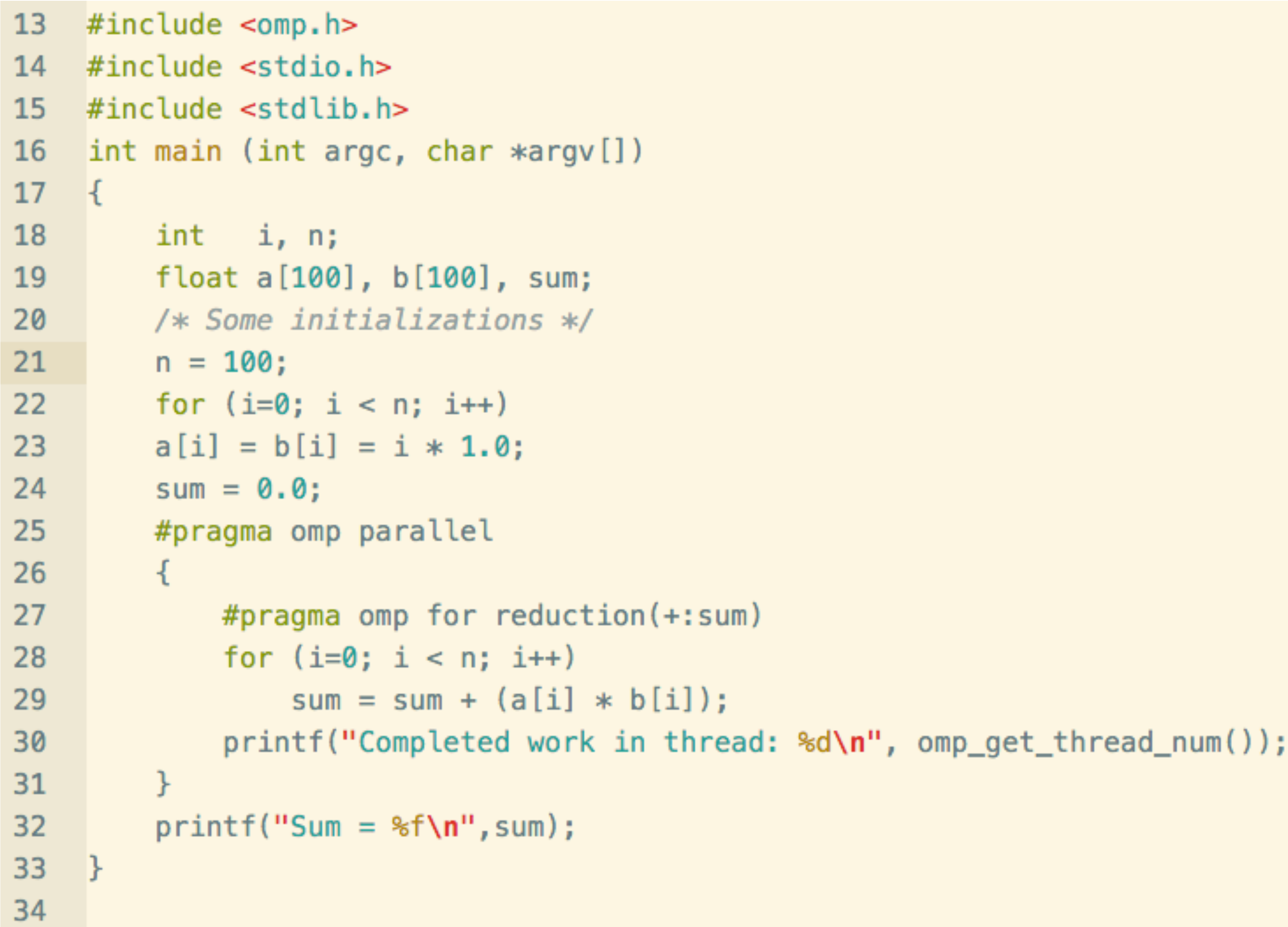
1. lastprivate (list)
2. reduction
3. schedule (list)
4. ordered
5. nowait

lastprivate (list)

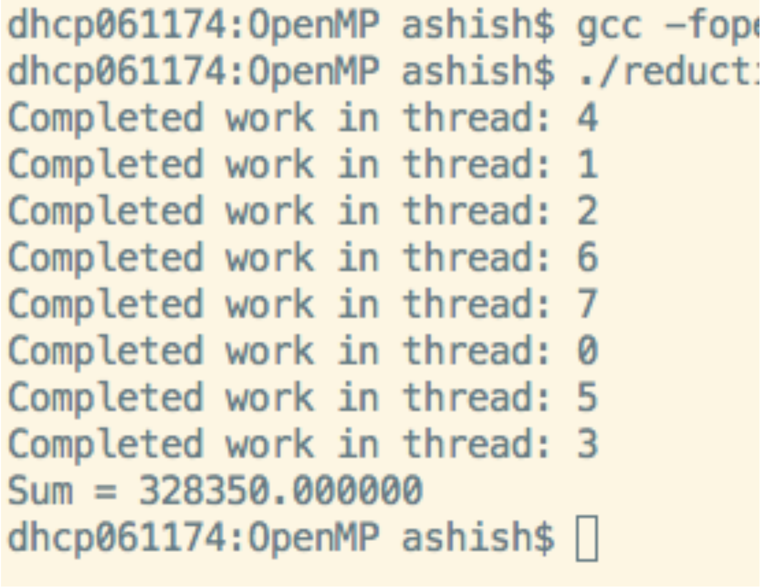
Similar to private

Value is available outside the loop
Value → What would be at the end of the loop, if code was serial

Could be ‘ready’, but not accessible as other threads maybe busy



```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[])
{
    int   i, n;
    float a[100], b[100], sum;
    /* Some initializations */
    n = 100;
    for (i=0; i < n; i++)
    a[i] = b[i] = i * 1.0;
    sum = 0.0;
    #pragma omp parallel
    {
        #pragma omp for reduction(+:sum)
        for (i=0; i < n; i++)
            sum = sum + (a[i] * b[i]);
        printf("Completed work in thread: %d\n", omp_get_thread_num());
    }
    printf("Sum = %f\n",sum);
}

```

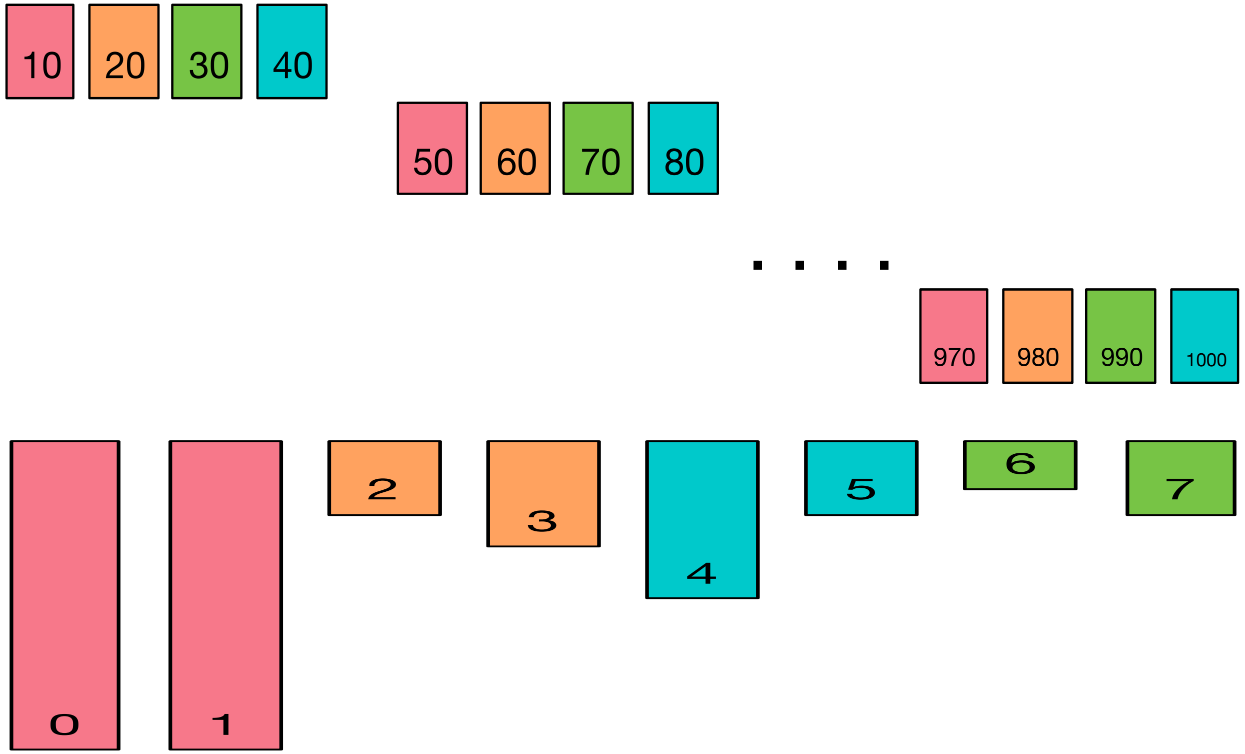


```
dhcp061174:OpenMP ashish$ gcc -fope
dhcp061174:OpenMP ashish$ ./reduct
Completed work in thread: 4
Completed work in thread: 1
Completed work in thread: 2
Completed work in thread: 6
Completed work in thread: 7
Completed work in thread: 0
Completed work in thread: 5
Completed work in thread: 3
Sum = 328350.000000
dhcp061174:OpenMP ashish$ 
```

Reduction clause: In a for loop   (see reduction.c)

Static Scheduling: #pragma omp parallel for schedule(static, 10)

- OpenMP does not specify how loops are partitioned, the typical implementation is one where a loop is split equally across the number of threads
- Chunk size, if specified, indicates the number of iterations
  → Size = 1000;
  4 threads; Chunk = 10
- Pros: Easy
- Cons → zz….. (load imbalance)



Runtime, Dynamic and Guided Scheduling

- runtime → specify using ENV variable (OMP_SCHEDULE)

- #pragma omp parallel for schedule (dynamic, n)
  - Chunk size of n (default 1)
  - When thread finishes one chunk, it is assigned a new one

- #pragma omp parallel for schedule (guided, n)
  - Chunk size is relative to the num of iterations left
  - Iterations are dynamically assigned to threads in blocks as threads request them
  - Similar to dynamic except that the block size decreases each time a parcel of work is given to a thread.
    - The size of the initial block is proportional to: #iterations / #threads
    - Subsequent blocks are proportional to: #iterations_remaining / #threads
  - The chunk parameter defines the minimum block size. The default chunk size is 1.

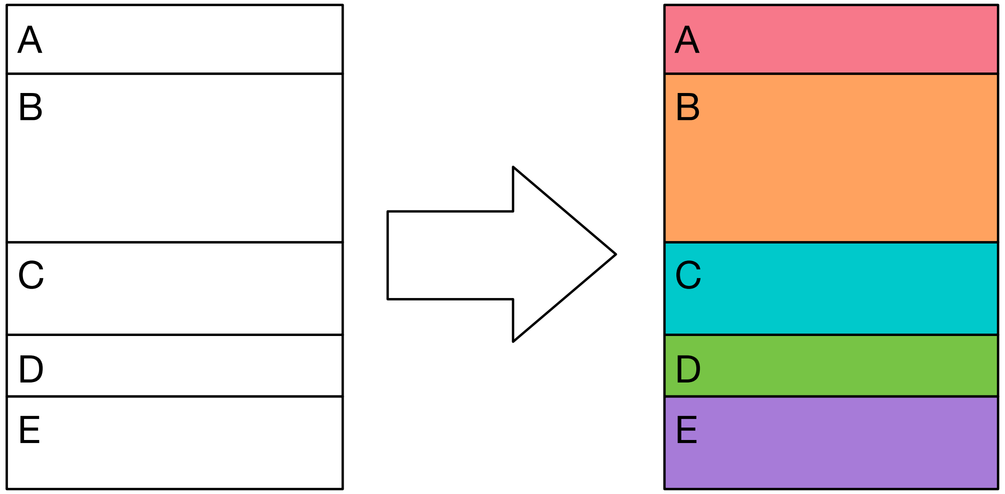
nowait

Don't wait for all threads to finish,
you can proceed

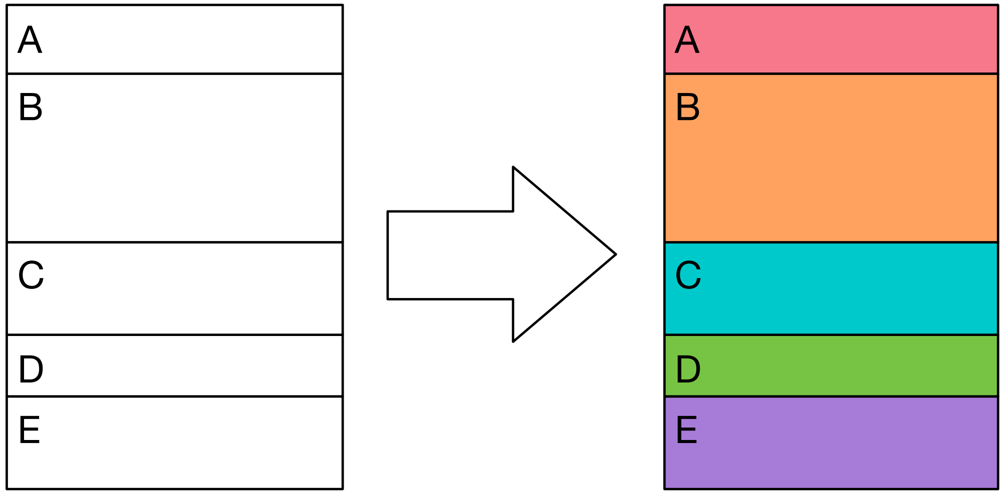
```
#pragma omp parallel
{
    #pragma omp for nowait
        for (i=1; i<n; i++)
            b[i] = (a[i] + a[i-1]) / 2.0;

    #pragma omp for nowait
        for (i=0; i<m; i++)
            y[i] = sqrt(z[i]);
}
```

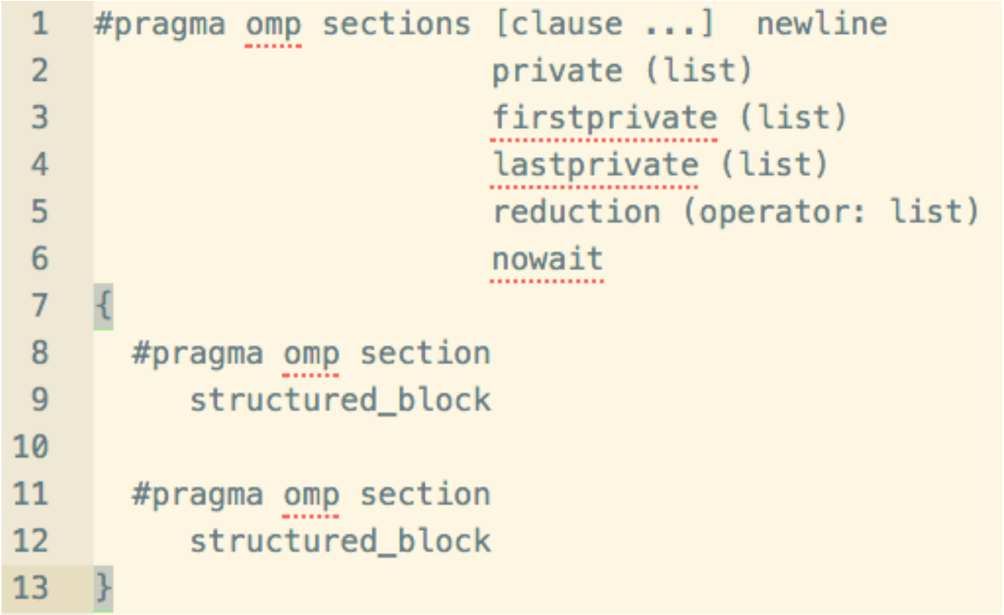
Sections

a non-iterative work-sharing construct



- Independent SECTION directives are nested within a SECTIONS directive.
- Each SECTION is executed once by a thread in the team.
- Different sections may be executed by different threads.
- It is possible for a thread to execute more than one section if it is quick enough and the implementation permits such.





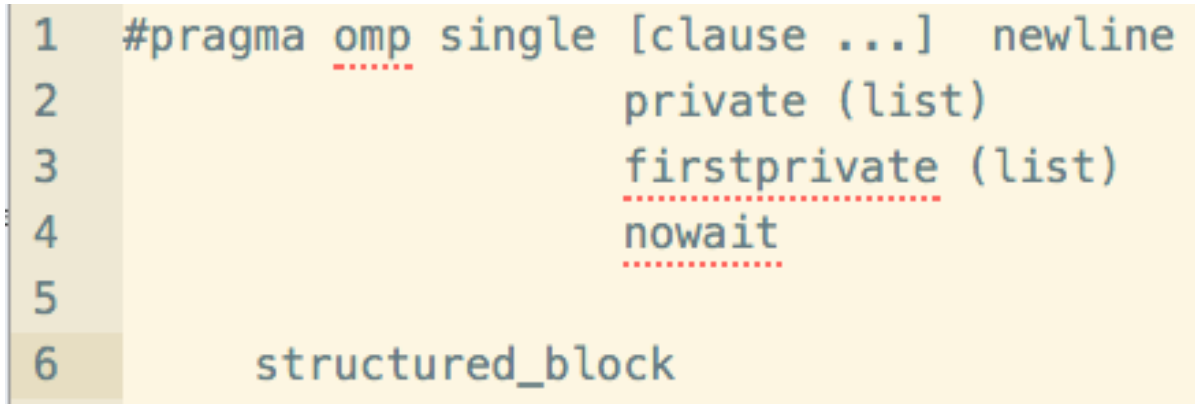
```
#pragma omp sections [clause ...]  newline
                     private (list)
                     firstprivate (list)
                     lastprivate (list)
                     reduction (operator: list)
                     nowait
{
  #pragma omp section
     structured_block

  #pragma omp section
     structured_block
}
```

Sections

a non-iterative work-sharing construct

Single

Only one thread will execute this



```
#pragma omp single [clause ...]  newline
                   private (list)
                   firstprivate (list)
                   nowait

     structured_block
```

Other threads will wait
Useful for thread-unsafe code
Useful for I/O operations

Synchronization

- OpenMP programs use shared variables to communicate. We need to make sure these variables are not accessed at the same time by different threads — Why?
- Available Directives:
  - Master
  - Critical
  - Atomic
  - Barrier

Master → #pragma omp master

- This Directive ensures that only the master threads executes instructions in the block. There is no implicit barrier so other threads will not wait for master to finish

Critical → #pragma omp critical

- Specifies a region of code that must be executed by only one thread at a time.
- If another threads reaches the critical section it will wait untill the current thread finishes this critical section.
- Every thread will execute the critical block and they will synchronize at end of critical section

```
#include <omp.h>
main()
{
int x;
x = 0;
#pragma omp parallel shared(x) {
  #pragma omp critical
       x = x + 1;
  }  /* end of parallel section */
}
```

Barrier

#pragma omp barrier

A barrier will force every thread to wait at the barrier until all threads have reached the barrier.

Atomic

#pragma omp atomic

This Directive is very similar to the CRITICAL directive on the previous slide. Difference is that ATOMIC is only used for the update of a memory location.

Write a simple matrix multiplier

C[N][1] = A[N][N] * B[N][1]

1. Initialize as follows:

```
srand((int)time(NULL));
X = rand() % 10
```

2. Surround timers around the loop construct

```
Start = omp_get_wtime();
       {C = A x B}
End = omp_get_wtime();
printf("---- Parallel done in %f seconds.\n", End - Start);
```

3. Run for N = 100000