# CISC 372: Parallel Computing OpenMP, Part 3

Stephen F. Siegel

Department of Computer and Information Sciences University of Delaware

・ロト ・日ト ・ヨト ・ヨト ・ヨー うへで

# OpenMP worksharing directives

Recall:

- used to divide up work among threads
- kinds of work-sharing constructs
  - for loops: distribute iterations to team members
  - sections: distribute independent code bocks (work units)
  - single: let only one thread execute a block

We left off looking at different clauses that can be used with the omp for directive.

this is another clause that can be added to an omp for directive

- this is another clause that can be added to an omp for directive
- > performs an (approximately) associative and commutative operation across all threads

- ▶ this is another clause that can be added to an omp for directive
- > performs an (approximately) associative and commutative operation across all threads
- $\blacktriangleright$  each variable v in the list should be a shared variable

- this is another clause that can be added to an omp for directive
- performs an (approximately) associative and commutative operation across all threads
- $\triangleright$  each variable v in the list should be a shared variable
- $\triangleright$  v should be initialized before entering the loop

- ▶ this is another clause that can be added to an omp for directive
- > performs an (approximately) associative and commutative operation across all threads
- $\blacktriangleright$  each variable v in the list should be a shared variable
- v should be initialized before entering the loop
- effectively, a private copy of v is created

- this is another clause that can be added to an omp for directive
- > performs an (approximately) associative and commutative operation across all threads
- $\blacktriangleright$  each variable v in the list should be a shared variable
- v should be initialized before entering the loop
- effectively, a private copy of v is created
- $\blacktriangleright$  each private v is initialized to the default initial value corresponding to the operation
  - ▶ 0 for +, 1 for \*, etc.

- this is another clause that can be added to an omp for directive
- > performs an (approximately) associative and commutative operation across all threads
- $\blacktriangleright$  each variable v in the list should be a shared variable
- v should be initialized before entering the loop
- effectively, a private copy of v is created
- $\blacktriangleright$  each private v is initialized to the default initial value corresponding to the operation
  - $\blacktriangleright$  0 for +, 1 for \*, etc.
- all operations in loop body take place on the private copies

- this is another clause that can be added to an omp for directive
- > performs an (approximately) associative and commutative operation across all threads
- $\blacktriangleright$  each variable v in the list should be a shared variable
- v should be initialized before entering the loop
- effectively, a private copy of v is created
- $\blacktriangleright$  each private v is initialized to the default initial value corresponding to the operation

3

- $\blacktriangleright$  0 for +, 1 for \*, etc.
- all operations in loop body take place on the private copies
- when a thread finishes its iterations:
  - $\blacktriangleright$  it adds (or whatever the operation is) its private value back to the shared v
  - this happens atomically to prevent races

### Reduction example: reduce.c

S.F. Siegel

0

```
#include <stdio h>
    #include <omp.h>
    #define n 10
    int a[n], s=1000000;
    int main() {
      printf("Start s = %d\n", s);
    #pragma omp parallel default(none) shared(a,s)
        int tid = omp_get_thread_num();
    #pragma omp for
        for (int i=0; i<n; i++) a[i] = i;
    #pragma omp for reduction(+:s) schedule(static,1)
        for (int i=0; i<n; i++) {</pre>
          s+=a[i]:
          printf("Local s on thread %d = %d n", tid, s);
        }
      printf("Final s = (d n), s);
    }
CISC 372: Parallel Computing
```

♦ OpenMP part 3

4

#### Reduction example: output

```
omp$ make reduce
cc -fopenmp -o reduce.exec reduce.c
./reduce.exec
Start s = 1000000
Local s on thread 0 = 0
Local s on thread 0 = 2
Local s on thread 0 = 6
Local s on thread 0 = 12
Local s on thread 0 = 20
Local s on thread 1 = 1
Local s on thread 1 = 4
Local s on thread 1 = 9
Local s on thread 1 = 16
Local s on thread 1 = 25
Final s = 1000045
omp$
```

S.F. Siegel  $\diamond$  CISC 372: Parallel Computing  $\diamond$  OpenMP part 3

## **Reduction operations**

operation	operator	initial value
addition	+	0
multiplication	*	1
subtraction (?)	-	0
bitwise and	&	~0
bitwise or	1	0
bitwise exclusive or	^	0
logical and	&&	1
logical or	11	0

S.F. Siegel  $\diamond$  CISC 372: Parallel Computing  $\diamond$  OpenMP part 3

7

◆□▶ ◆□▶ ◆豆▶ ◆豆▶ = うへぐ

▶ iterations are partitioned into chunks of size *chunk\_size* 

S.F. Siegel  $\diamond$  CISC 372: Parallel Computing  $\diamond$  OpenMP part 3

◆□▶ ◆□▶ ◆豆▶ ◆豆▶ = うへぐ

- ▶ iterations are partitioned into chunks of size *chunk\_size*
- chunks are distributed in round-robin order to threads

- ▶ iterations are partitioned into chunks of size *chunk\_size*
- chunks are distributed in round-robin order to threads
- last chunk may be smaller

- iterations are partitioned into chunks of size chunk\_size
- chunks are distributed in round-robin order to threads
- last chunk may be smaller
- distribution is "static": determined upon reaching the loop

- iterations are partitioned into chunks of size chunk\_size
- chunks are distributed in round-robin order to threads
- last chunk may be smaller
- distribution is "static": determined upon reaching the loop
- you can omit chunk\_size
  - iteration space divided into chunks of approximately equal size
  - at most one chunk given to each thread

S.F. Siegel  $\diamond$  CISC 372: Parallel Computing  $\diamond$  OpenMP part 3

- イロト イボト イモト モー シベウ

iterations are partitioned into chunks of size chunk\_size

- iterations are partitioned into chunks of size chunk\_size
- chunks are distributed to threads as they request them
  - similar to the "manager-worker" pattern
  - as soon as a thread completes its chunk, it asks for a new one

- iterations are partitioned into chunks of size chunk\_size
- chunks are distributed to threads as they request them
  - similar to the "manager-worker" pattern
  - as soon as a thread completes its chunk, it asks for a new one
- last chunk may be smaller

- iterations are partitioned into chunks of size chunk\_size
- chunks are distributed to threads as they request them
  - similar to the "manager-worker" pattern
  - as soon as a thread completes its chunk, it asks for a new one
- last chunk may be smaller
- > advantageous when time to execute an iteration varies in an unpredictable way

- iterations are partitioned into chunks of size chunk\_size
- chunks are distributed to threads as they request them
  - similar to the "manager-worker" pattern
  - as soon as a thread completes its chunk, it asks for a new one
- last chunk may be smaller
- > advantageous when time to execute an iteration varies in an unpredictable way
- distribution is "dynamic": determined as loop executes

S.F. Siegel  $\diamond$  CISC 372: Parallel Computing  $\diamond$  OpenMP part 3

9

- \* ロ > \* 個 > \* 画 > \* 画 > - 画 - の < @

▶ this is a variation on *dynamic* in which the chunk size decreases as execution proceeds

- this is a variation on dynamic in which the chunk size decreases as execution proceeds
- size of chunk proportional to number of unassigned iterations divided by number of threads

9

♦ OpenMP part 3

- this is a variation on dynamic in which the chunk size decreases as execution proceeds
- size of chunk proportional to number of unassigned iterations divided by number of threads
  - chunk\_size is a lower bound on the size of a chunk
  - for  $chunk\_size = 1$ , size of a chunk decreases to 1
  - for  $chunk_size = k > 1$ , all chunks other than last must contain at least k iterations

- this is a variation on dynamic in which the chunk size decreases as execution proceeds
- size of chunk proportional to number of unassigned iterations divided by number of threads
  - chunk size is a lower bound on the size of a chunk
  - for *chunk* size = 1, size of a chunk decreases to 1
  - **•** for  $chunk\_size = k > 1$ , all chunks other than last must contain at least k iterations

#### motivation

there is overhead to the manager-worker protocol

- ▶ this is a variation on *dynamic* in which the chunk size decreases as execution proceeds
- size of chunk proportional to number of unassigned iterations divided by number of threads
  - chunk\_size is a lower bound on the size of a chunk
  - for  $chunk\_size = 1$ , size of a chunk decreases to 1
  - for  $chunk_size = k > 1$ , all chunks other than last must contain at least k iterations

#### motivation

- there is overhead to the manager-worker protocol
- $\blacktriangleright$  bigger chunks  $\rightarrow$  less overhead, but greater probability of leaving a thread idle

- ▶ this is a variation on *dynamic* in which the chunk size decreases as execution proceeds
- size of chunk proportional to number of unassigned iterations divided by number of threads
  - chunk\_size is a lower bound on the size of a chunk
  - for  $chunk\_size = 1$ , size of a chunk decreases to 1
  - for  $chunk_size = k > 1$ , all chunks other than last must contain at least k iterations

#### motivation

- there is overhead to the manager-worker protocol
- $\blacktriangleright$  bigger chunks  $\rightarrow$  less overhead, but greater probability of leaving a thread idle
- compromise: increase granularity as iteration space gets smaller, when the chance of leaving a thread idle is greater

To wait or not to wait?

## To wait or not to wait?

use of nowait clause in for directive removes the implicit barrier at end of loop

## To wait or not to wait?

- use of nowait clause in for directive removes the implicit barrier at end of loop
- this can increase concurrency, and performance

# To wait or not to wait?

- use of nowait clause in for directive removes the implicit barrier at end of loop
- this can increase concurrency, and performance
- but can also introduce bugs
  - use with extreme caution
  - make sure it does not introduce data races

# To wait or not to wait: wait1.c

```
int main () {
  double a[n]. b[n]:
#pragma omp parallel default(none) shared(a,b)
  Ł
#pragma omp for nowait
    for (int i=0; i<n; i++)</pre>
      a[i] = 2.0*i:
#pragma omp for
    for (int i=0; i<n; i++)</pre>
      b[i] = 3.0*i:
 } /* end of parallel region */
  for (int i=0; i<n; i++) {</pre>
    if (a[i]!=2.0*i) { printf("Error at a[%d]: %f\n", i, a[i]); fflush(stdout); exit(1); }
    if (b[i]!=3.0*i) { printf("Error at b[%d]: %f\n", i, b[i]); fflush(stdout); exit(1); }
  }
  printf("Success\n");
}
```

# To wait or not to wait: wait1.c

```
int main () {
  double a[n]. b[n]:
#pragma omp parallel default(none) shared(a,b)
  Ł
#pragma omp for nowait
    for (int i=0; i<n; i++)</pre>
      a[i] = 2.0*i:
#pragma omp for
    for (int i=0; i<n; i++)</pre>
      b[i] = 3.0*i:
 } /* end of parallel region */
  for (int i=0; i<n; i++) {</pre>
    if (a[i]!=2.0*i) { printf("Error at a[%d]: %f\n", i, a[i]); fflush(stdout); exit(1); }
    if (b[i]!=3.0*i) { printf("Error at b[%d]: %f\n", i, b[i]); fflush(stdout); exit(1); }
  }
  printf("Success\n");
}
```

OK: the two loops can execute concurrently since they update distinct variables

# To wait or not to wait: wait2.c

S.F. Siegel

0

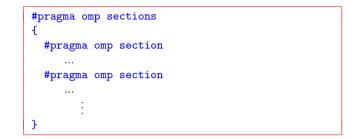
```
int main() {
  double a[n]. b[n]:
#pragma omp parallel default(none) shared(a,b)
#pragma omp for nowait
    for (int i=0; i<n; i++) a[i] = 2.0*i;
#pragma omp for
    for (int i=0; i<n; i++) b[i] = 2.0*a[n-i-1];
  } /* end of parallel region */
  for (int i=0: i<n: i++) {</pre>
    if (a[i] != 2.0*i) {
      printf("Error at a[%d]: %f\n", i, a[i]); fflush(stdout); exit(1);
    }
    if (b[i] != 2.0*(2.0*(n-i-1))) {
      printf("Error at b[%d]: %f\n", i, b[i]); fflush(stdout); exit(1);
  3
  printf("Success 2\n");
          CISC 372: Parallel Computing
                                ♦ OpenMP part 3
```

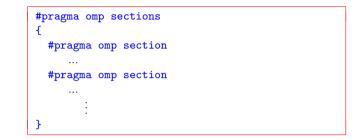
# To wait or not to wait: wait2.c

```
int main() {
  double a[n]. b[n]:
#pragma omp parallel default(none) shared(a,b)
#pragma omp for nowait
    for (int i=0; i<n; i++) a[i] = 2.0*i;
#pragma omp for
    for (int i=0; i<n; i++) b[i] = 2.0*a[n-i-1];
  } /* end of parallel region */
  for (int i=0: i<n: i++) {</pre>
    if (a[i] != 2.0*i) {
      printf("Error at a[%d]: %f\n", i, a[i]); fflush(stdout); exit(1);
    }
    if (b[i] != 2.0*(2.0*(n-i-1))) {
      printf("Error at b[%d]: %f\n", i, b[i]); fflush(stdout); exit(1);
  3
  printf("Success 2\n");
```

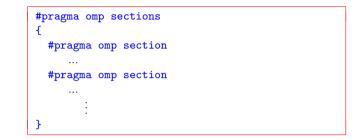
12

NOT OK: second loop reads variables assigned in the first loop. Run it, and then run wait2\_fix.c.

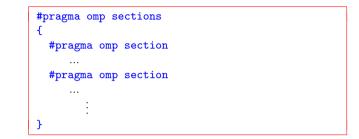




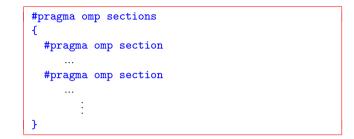
specifies explicit code blocks which can execute in parallel



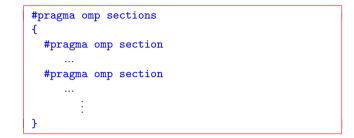
- specifies explicit code blocks which can execute in parallel
- each block (or section) is executed once, by exactly one thread



- specifies explicit code blocks which can execute in parallel
- each block (or section) is executed once, by exactly one thread
- a thread may execute several sections, or no sections



- specifies explicit code blocks which can execute in parallel
- each block (or section) is executed once, by exactly one thread
- a thread may execute several sections, or no sections
- ▶ in general: you cannot assume anything about how sections are distributed to threads



- specifies explicit code blocks which can execute in parallel
- each block (or section) is executed once, by exactly one thread
- a thread may execute several sections, or no sections
- ▶ in general: you cannot assume anything about how sections are distributed to threads
- barrier at end (unless overridden with nowait)

#### sections example: sections.c, part 1

```
#include <stdio.h>
#include <omp.h>
#include <limits.h>
#define N 20
typedef unsigned long ulong;
ulong sumUpTo(int n) {
  ulong s=0;
  for (int i=1; i<=n; i++) s+=i;</pre>
  return s;
}
ulong productUpTo(int n) {
  ulong p=1;
  for (int i=1; i<=n; i++) p*=i;</pre>
  return p;
}
```

#### sections example: sections.c, part 2

```
int main() {
#pragma omp parallel
 { /* begin parallel region */
   int tid = omp_get_thread_num();
   if (tid == 0) printf("Number of threads: %d\n", omp_get_num_threads());
#pragma omp sections
   { /* begin sections */
#pragma omp section
       printf("Thread %d: sum to %d ..... %lu\n", tid, N, sumUpTo(N));
#pragma omp section
       printf("Thread %d: product to %d ...... %lu\n", tid, N, productUpTo(N));
      3
   } /* end of sections */
 } /* end of parallel region */
}
```

# Clauses allowed with sections

- private(list)
  - each section has its own private copy of variable
- firstprivate(list)
  - make private and initialize with shared variable value
- lastprivate(list)
  - value of private copy of variable in last section is copied to shared variable at end
- reduction(reduction-identifier:list)
  - reduction applied across all sections
- nowait
  - removes barrier at end

#pragma omp single S

```
#pragma omp single
S
```

indicates that you want only one thread in the team to execute S

you don't care which thread

```
#pragma omp single
S
```

 $\blacktriangleright$  indicates that you want only one thread in the team to execute S

you don't care which thread

barrier at end (unless overridden with nowait)

17

♦ OpenMP part 3

```
#pragma omp single
S
```

 $\blacktriangleright$  indicates that you want only one thread in the team to execute S

- you don't care which thread
- barrier at end (unless overridden with nowait)
- typical use: initialization of shared variable

```
#pragma omp single
S
```

- $\blacktriangleright$  indicates that you want only one thread in the team to execute S
  - you don't care which thread
- barrier at end (unless overridden with nowait)
- typical use: initialization of shared variable

Clauses:

private(list), firstprivate(list), nowait: usual semantics

```
#pragma omp single
S
```

- indicates that you want only one thread in the team to execute S
  - you don't care which thread
- barrier at end (unless overridden with nowait)
- typical use: initialization of shared variable

Clauses:

- private(list), firstprivate(list), nowait: usual semantics
- copyprivate(list)
  - applies to private variables
  - copies final value of variable in the single thread to corresponding variables in all other threads

17

copy occurs at end, before threads leave the barrier