COMP 605: Introduction to Parallel Computing
Topic: OpenMP Parallel For Directives

Mary Thomas

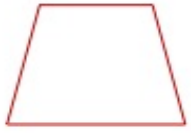
Department of Computer Science
Computational Science Research Center (CSRC)
San Diego State University (SDSU)

Presented: 04/13/17
Last Update: 04/10/17

Parallel for

- Forks a team of threads to execute the following structured block.
- However, the structured block following the parallel for directive must be a for loop.
- Furthermore, with the parallel for directive the system parallelizes the for loop by dividing the iterations of the loop among the threads.

```
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++)
   approx += f(a + i*h);
approx = h*approx;
```



```
   h = (b-a)/n;
   approx = (f(a) + f(b))/2.0;
#  pragma omp parallel for num_threads(thread_count) \
      reduction(+: approx)
   for (i = 1; i <= n-1; i++)
      approx += f(a + i*h);
   approx = h*approx;
```


Good News/Bad News: threads divide up work themselves (Good), but you cannot control this (maybe Bad). Not useful when you need/want to avoid default data composition.

Legal forms for parallelizable for statements

```
for ( index = start ; index < end    ; index++ )
                    ; index <= end   ; ++index
                    ; index >= end   ; index--
                    ; index > end    ; --index
                                     ; index += incr
                                     ; index -= incr
                                     ; index = index + incr
                                     ; index = incr + index
                                     ; index = index - incr )
```

Caveats

- The variable index must have integer or pointer type (e.g., it can't be a float).
- The expressions start, end, and incr must have a compatible type. For example, if index is a pointer, then incr must have integer type.

Caveats

- The expressions start, end, and incr must not change during execution of the loop.
- During execution of the loop, the variable index can only be modified by the "increment expression" in the for statement.

Data dependencies

```
fibo[ 0 ] = fibo[ 1 ] = 1;
for (i = 2; i < n; i++)
    fibo[ i ] = fibo[ i – 1 ] + fibo[ i – 2 ];
```

note 2 threads

```
fibo[ 0 ] = fibo[ 1 ] = 1;
# pragma omp parallel for num_threads(2)
for (i = 2; i < n; i++)
    fibo[ i ] = fibo[ i – 1 ] + fibo[ i – 2 ];
```

1 1 2 3 5 8 13 21 34 55

this is correct

but sometimes we get this

1 1 2 3 5 8 0 0 0 0

Pacheco Data Dependencies example: omp_fibo.c

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

int main(int argc, char* argv[]) {
   int thread_count, n, i;
   long long* fibo;

   if (argc != 3) Usage(argv[0]);
   thread_count = strtol(argv[1], NULL, 10);
   n = strtol(argv[2], NULL, 10);

   fibo = malloc(n*sizeof(long long));
   fibo[0] = fibo[1] = 1;
#  pragma omp parallel for num_threads(thread_count)
   for (i = 2; i < n; i++)
      fibo[i] = fibo[i-1] + fibo[i-2];

   printf("The first n Fibonacci numbers:\n");
   for (i = 0; i < n; i++)
      printf("%d\t%lld\n", i, fibo[i]);

   free(fibo);
   return 0;
}  /* main */
```

Pacheco Data Dependencies example: local laptop had thread problems, but this did not happen on tuckoo (up to 10,000 threads, no problems)

```
[gidget:intro-par-pgming-pacheco/ipp-source/ch5] mthomas% gcc -fopenmp -o omp_fibo omp_fibo.c
[gidget:intro-par-pgming-pacheco/ipp-source/ch5] mthomas% ./omp_fibo 10 10
The first n Fibonacci numbers:
0       1
1       1
2       2
3       1
4       3
5       1
6       0
7       0
8       0
9       0
[gidget:intro-par-pgming-pacheco/ipp-source/ch5] mthomas% ./omp_fibo 10 10
The first n Fibonacci numbers:
0       1
1       1
2       2
3       1
4       3
5       4
6       7
7       0
8       0
9       0
[gidget:intro-par-pgming-pacheco/ipp-source/ch5] mthomas%
```

What went wrong?

- OpenMP compilers dont check for dependences among iterations in a loop thats being parallelized with a parallel for directive.
- A loop in which the results of one or more iterations depend on other iterations cannot, in general, be correctly parallelized by OpenMP.
- Dependencies within the current loop is OK:
  #pragma omp parallel *for* *numthreads* (*threadcount*)

$$x[i] = a + i * h$$
$$y[i] = exp(x[i])$$

Estimating π

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right] = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}$$

```
double factor = 1.0;
double sum = 0.0;
for (k = 0; k < n; k++) {
    sum += factor/(2*k+1);
    factor = -factor;
}
pi_approx = 4.0*sum;
```

OpenMP solution #1

loop dependency

```
        double factor = 1.0;
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           reduction(+:sum)
        for (k = 0; k < n; k++) {
           sum += factor/(2*k+1);
           factor = -factor;
        }
        pi_approx = 4.0*sum;
```

Examine the series: for *pi*:

$$4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}$$

Replace $(-1)^k$

```
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
            default(none) reduction(+:sum) private(k, factor) \
            shared(n)
        for (k = 0; k < n; k++) {
            if (k % 2 == 0)
               factor = 1.0;
            else
               factor = -1.0;
            sum += factor/(2*k+1);
        }
```

Pacheco Data Dependencies example: omp_pi.c

```
int main(int argc, char* argv[]) {
   long long n, i;
   int thread_count;
   double factor;
   double sum = 0.0;

   if (argc != 3) Usage(argv[0]);
   thread_count = strtol(argv[1], NULL, 10);
   n = strtoll(argv[2], NULL, 10);
   if (thread_count < 1 || n < 1) Usage(argv[0]);

#  pragma omp parallel for num_threads(thread_count) \
      reduction(+: sum) private(factor)
   for (i = 0; i < n; i++) {
      factor = (i % 2 == 0) ? 1.0 : -1.0;
      sum += factor/(2*i+1);
#     ifdef DEBUG
      printf("Thread %d > i = %lld, my_sum = %f\n", my_rank, i, my_sum);
#     endif
   }

   sum = 4.0*sum;
   printf("With n = %lld terms and %d threads,\n", n, thread_count);
   printf("   Our estimate of pi = %.14f\n", sum);
   printf("                      pi = %.14f\n", 4.0*atan(1.0));
   return 0;
}  /* main */
```

Running omp_pi.c: could reproduce erroneous condition on OS Mountain Lion - not every time

```
With n = 100 terms and 10 threads,
   Our estimate of pi = 3.13159290355855
                pi = 3.14159265358979
[gidget:intro-par-pgming-pacheco/ipp-source/ch5] mthomas% ./omp_pi 1 1000
With n = 1000 terms and 1 threads,
   Our estimate of pi = 3.14059265383979
                pi = 3.14159265358979
[gidget:intro-par-pgming-pacheco/ipp-source/ch5] mthomas% ./omp_pi-noprivatfactor 2 1000
With n = 1000 terms and 2 threads,
   Our estimate of pi = 3.14803451430491
                pi = 3.14159265358979
dget:intro-par-pgming-pacheco/ipp-source/ch5] mthomas% ./omp_pi 10 1000000
With n = 1000000 terms and 10 threads,
   Our estimate of pi = 3.14159165358972
                pi = 3.14159265358979
[gidget:intro-par-pgming-pacheco/ipp-source/ch5] mthomas% ./omp_pi 10 10000000
With n = 10000000 terms and 10 threads,
   Our estimate of pi = 3.14159255358977
                pi = 3.14159265358979
```

The default clause

- Lets the programmer specify the scope of each variable in a block.

```
default(none)
```

- With this clause the compiler will require that we specify the scope of each variable we use in the block and that has been declared outside the block.

The default clause

```
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           default(none) reduction(+:sum) private(k, factor) \
           shared(n)
        for (k = 0; k < n; k++) {
           if (k % 2 == 0)
              factor = 1.0;
           else
              factor = -1.0;
           sum += factor/(2*k+1);
        }
```

Bubble Sort: $\vartheta\left(n^2\right)$

Bubble Sort

```
for (list_length = n; list_length >= 2; list_length--)
   for (i = 0; i < list_length-1; i++)
      if (a[i] > a[i+1]) {
         tmp = a[i];
         a[i] = a[i+1];
         a[i+1] = tmp;
      }
```

Loop carried dependence in outer loop - different threads sorting different data could result in wrong ordering.

Serial Odd-Even Transposition Sort

```
for (phase = 0; phase < n; phase++)
   if (phase % 2 == 0)
      for (i = 1; i < n; i += 2)
         if (a[i-1] > a[i]) Swap(&a[i-1],&a[i]);
   else
      for (i = 1; i < n-1; i += 2)
         if (a[i] > a[i+1]) Swap(&a[i], &a[i+1]);
```


See Pacheco, Chpt 3.7.

Odd-Even Transposition Sort: $\vartheta(n)$

Serial Odd-Even Transposition Sort

Phase	Subscript in Array 0		1		2		3
0	9	↔	7		8	↔	6
	7		9		6		8
1	7		9	↔	6		8
	7		6		9		8
2	7	↔	6		9	↔	8
	6		7		8		9
3	6		7	↔	8		9
	6		7		8		9

First OpenMP Odd-Even Sort

```
for (phase = 0; phase < n; phase++) {
   if (phase % 2 == 0)
#     pragma omp parallel for num_threads(thread_count) \
         default(none) shared(a, n) private(i, tmp)
      for (i = 1; i < n; i += 2) {
         if (a[i-1] > a[i]) {
            tmp = a[i-1];
            a[i-1] = a[i];
            a[i] = tmp;
         }
      }
   else
#     pragma omp parallel for num_threads(thread_count) \
         default(none) shared(a, n) private(i, tmp)
      for (i = 1; i < n-1; i += 2) {
         if (a[i] > a[i+1]) {
            tmp = a[i+1];
            a[i+1] = a[i];
            a[i] = tmp;
         }
      }
}
```


52

Issues: implicit barrier after each phase completes; overhead associated with fork/join.

Second OpenMP Odd-Even Sort

```
#  pragma omp parallel num_threads(thread_count) \
      default(none) shared(a, n) private(i, tmp, phase)
   for (phase = 0; phase < n; phase++) {
      if (phase % 2 == 0)
#        pragma omp for
         for (i = 1; i < n; i += 2) {
            if (a[i-1] > a[i]) {
               tmp = a[i-1];
               a[i-1] = a[i];
               a[i] = tmp;
            }
         }
      else
#        pragma omp for
         for (i = 1; i < n-1; i += 2) {
            if (a[i] > a[i+1]) {
               tmp = a[i+1];
               a[i+1] = a[i];
               a[i] = tmp;
            }
         }
   }
```

Odd-even sort with two parallel for directives and two for directives. (Times are in seconds.)

thread_count	1	2	3	4
Two `parallel` **for** directives	0.770	0.453	0.358	0.305
Two **for** directives	0.732	0.376	0.294	0.239

OpenMP assigns/distributes work (block partitioning)
E.g., for n iterations, then each thread gets $\frac{n}{thread\_count}$
Alternative approach: cyclic partitioning.

We want to parallelize this loop.

```
sum = 0.0;
for (i = 0; i <= n; i++)
   sum += f(i);
```

Thread	Iterations
0	$0, n/t, 2n/t, \ldots$
1	$1, n/t+1, 2n/t+1, \ldots$
$\vdots$	$\vdots$
$t-1$	$t-1, n/t+t-1, 2n/t+t-1, \ldots$

Assignment of work using cyclic partitioning.


```
double f(int i) {
   int j, start = i*(i+1)/2, finish = start + i;
   double return_val = 0.0;

   for (j = start; j <= finish; j++) {
      return_val += sin(j);
   }
   return return_val;
} /* f */
```

Our definition of function *f*.

Results

- f(i) calls the sin function *i* times.
- Assume the time to execute f(2i) requires approximately twice as much time as the time to execute f(i).

- n = 10,000
  - one thread
  - run-time = 3.67 seconds.

Results

- n = 10,000
  - two threads
  - default assignment
  - run-time = 2.76 seconds
  - speedup = 1.33
- n = 10,000
  - two threads
  - cyclic assignment
  - run-time = 1.84 seconds
  - speedup = 1.99

Scheduling: OpenMP assignment of threads.

The Schedule Clause

- Default schedule:

```
        sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           reduction(+:sum)
        for (i = 0; i <= n; i++)
           sum += f(i);
```

- Cyclic schedule:

```
        sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           reduction(+:sum) schedule(static,1)
        for (i = 0; i <= n; i++)
           sum += f(i);
```

Scheduling test code(Pacheco, 5.7, pg 238)

```
   double funtst(int);  /* Thread function */
/*--------------------------------------------------------------------*/
int main(int argc, char* argv[]) {
   int thread_count = strtol(argv[1], NULL, 10);
   int i, n;
   double sum;
   sum = 0.0;
   n=100000;
#  pragma omp parallel for num_threads(thread_count) \
      reduction(+:sum) schedule(static,1)
   for( i=0; i <=n; i++) {
      sum += funtst(i);
   }
   printf("N=%d, sum=%f\n",n,sum);
   return 0;
}  /* main */

/*--------------------------------------------------------------------
 * Function:    Hello
 * Purpose:     Thread function that prints message
 */
double funtst(int i) {
    int j, start=i*(i+1)/2, finish=start+i;
    for(j=start;j<=finish;j++) {
        ret_val += sin(j);
    }
    return ret_val;
}  /* funtst */
```

results

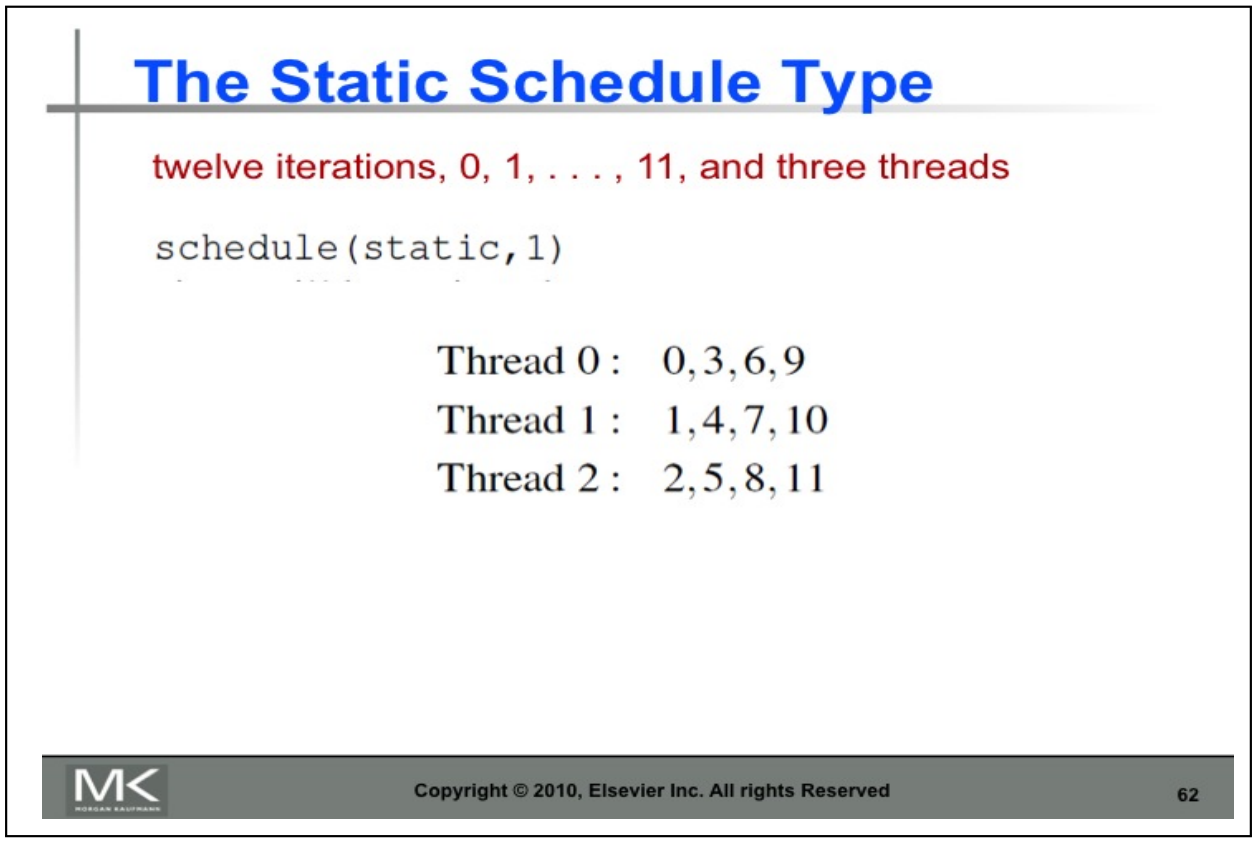
```
==== WITHOUT  SCHEDULING  ====================
[gidget]  mthomas% date ; ./omp_myt 1 ; date
Tue Nov  6 15:46:51 PST 2012
N=100000, sum=-0.284234
Tue Nov  6 15:50:18 PST 2012
[gidget]  mthomas% date ; ./omp_myt 2; date
Tue Nov  6 15:51:08 PST 2012
N=100000, sum=-1.027596
Tue Nov  6 15:52:53 PST 2012
[gidget]  mthomas% date ; ./omp_myt 8 ; date
Tue Nov  6 15:46:12 PST 2012
N=100000, sum=-1.931278
Tue Nov  6 15:46:40 PST 2012
[gidget] mthomas% date ; ./omp_myt 16 ; date
Tue Nov  6 15:42:40 PST 2012
N=100000, sum=7.887650
Tue Nov  6 15:43:12 PST 2012
[gidget] mthomas% date ; ./omp_myt 32 ; date
Tue Nov  6 15:43:56 PST 2012
N=100000, sum=22.965623
Tue Nov  6 15:44:27 PST 2012
```

```
==== WITH  SCHEDULING  ====================
[gidget]mthomas% date ; ./omp_myt_sch 1 ; date
Tue Nov  6 15:33:20 PST 2012
N=100000, sum=-0.284234
Tue Nov  6 15:37:05 PST 2012
[gidget] mthomas% date ; ./omp_myt_sch 2 ; date
Tue Nov  6 15:42:24 PST 2012
N=100000, sum=-0.284234
Tue Nov  6 15:44:51 PST 2012
[gidget] mthomas% date ; ./omp_myt_sch 8 ; date
Tue Nov  6 15:32:43 PST 2012
N=100000, sum=-0.284234
Tue Nov  6 15:33:12 PST 2012
[gidget]  mthomas% date ; ./omp_myt_sch 16 ; date
Tue Nov  6 15:32:05 PST 2012
N=100000, sum=-0.284234
Tue Nov  6 15:32:34 PST 2012
[gidget] mthomas% date; ./omp_myt_sch 32; date
Tue Nov  6 15:31:11 PST 2012
N=100000, sum=-0.284234
Tue Nov  6 15:31:40 PST 2012
```

schedule (type , chunksize)

- Type can be:
  - static: the iterations can be assigned to the threads before the loop is executed.
  - dynamic or guided: the iterations are assigned to the threads while the loop is executing.
  - auto: the compiler and/or the run-time system determine the schedule.
  - runtime: the schedule is determined at run-time.
- The chunksize is a positive integer.

chunks assigned in round-robin fashion.



The Static Schedule Type

twelve iterations, 0, 1, . . . , 11, and three threads

```
schedule(static,2)
```

Thread 0 : 0, 1, 6, 7
Thread 1 : 2, 3, 8, 9
Thread 2 : 4, 5, 10, 11

The Static Schedule Type

twelve iterations, 0, 1, . . . , 11, and three threads

```
schedule(static,4)
```

Thread 0 : 0, 1, 2, 3
Thread 1 : 4, 5, 6, 7
Thread 2 : 8, 9, 10, 11

The Dynamic Schedule Type

- The iterations are also broken up into chunks of chunksize consecutive iterations.
- Each thread executes a chunk, and when a thread finishes a chunk, it requests another one from the run-time system.
- This continues until all the iterations are completed.
- The chunksize can be omitted. When it is omitted, a chunksize of 1 is used.

The Guided Schedule Type

- Each thread also executes a chunk, and when a thread finishes a chunk, it requests another one.
- However, in a guided schedule, as chunks are completed the size of the new chunks decreases.
- If no chunksize is specified, the size of the chunks decreases down to 1.
- If chunksize is specified, it decreases down to chunksize, with the exception that the very last chunk can be smaller than chunksize.

Thread	Chunk	Size of Chunk	Remaining Iterations
0	1 – 5000	5000	4999
1	5001 – 7500	2500	2499
1	7501 – 8750	1250	1249
1	8751 – 9375	625	624
0	9376 – 9687	312	312
1	9688 – 9843	156	156
0	9844 – 9921	78	78
1	9922 – 9960	39	39
1	9961 – 9980	20	19
1	9981 – 9990	10	9
1	9991 – 9995	5	4
0	9996 – 9997	2	2
1	9998 – 9998	1	1
0	9999 – 9999	1	0

Assignment of trapezoidal rule iterations 1–9999 using a guided schedule with two threads.

The Runtime Schedule Type

- The system uses the environment variable OMP_SCHEDULE to determine at run-time how to schedule the loop.
- The OMP_SCHEDULE environment variable can take on any of the values that can be used for a static, dynamic, or guided schedule.