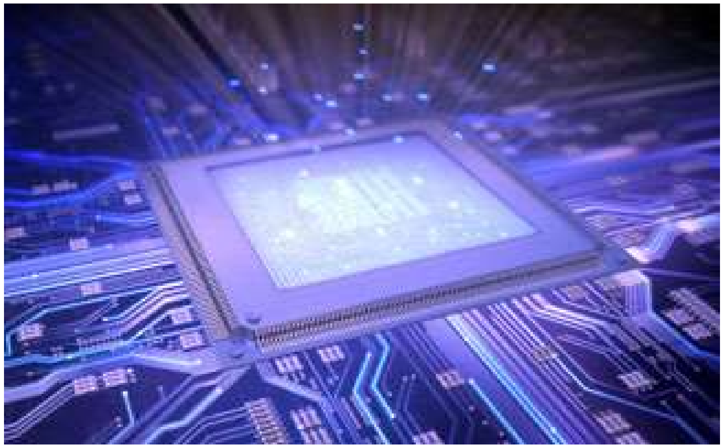
Parallel Computing

OpenMP III

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com



MORE ABOUT LOOPS IN OPENMP: SORTING

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 dependency in inner loop

Loop-carried dependency in outer loop

What can we do?

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]);
```

Serial Odd-Even Transposition Sort

Phase	Subscript in Array 0		1		2		3
0	9	$\leftrightarrow$	7		8	$\leftrightarrow$	6
	7		9		6		8
1	7		9	$\leftrightarrow$	6		8
	7		6		9		8
2	7	$\leftrightarrow$	6		9	$\leftrightarrow$	8
	6		7		8		9
3	6		7	$\leftrightarrow$	8		9
	6		7		8		9

Serial Odd-Even Transposition Sort

No dependence in inner loops

```
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]);
```

Outer-loop carried dependence

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;
                }
            }
    }
```

What if a thread proceeds from phase p to phase p+1 before other threads?

Performance issue:
For each outer iteration, OpenMP will fork-join threads → Repeated overhead per iteration.
Can we do better?

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;
             }
          }
    }
```

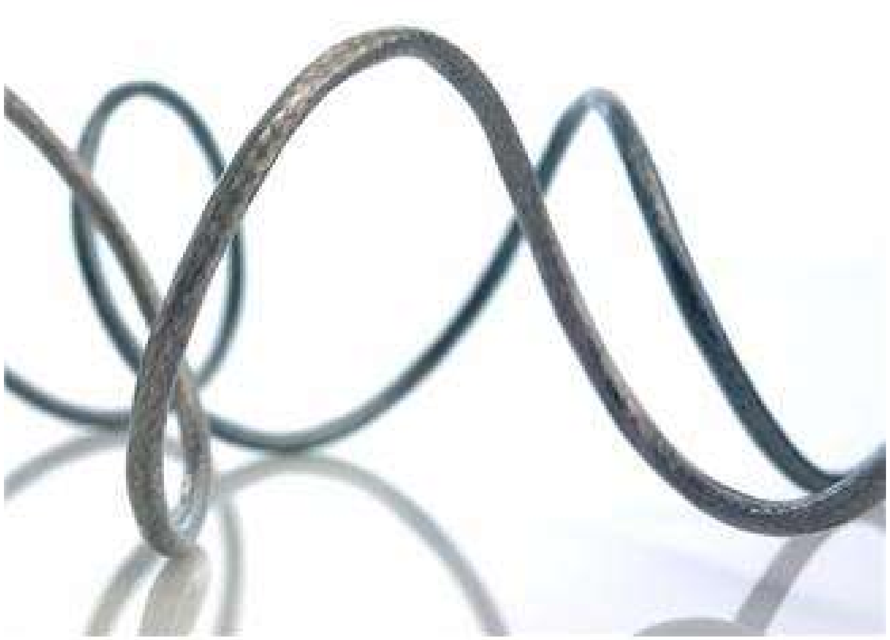
for directive does not fork any threads. But uses whatever threads that have been forked before in the enclosing parallel block.

Odd-even sort with two parallel for directives and two for directives.

(Times are in seconds.)

Array of 20,000 elements

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



SCHEDULING LOOPS

Take a look at this:

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

- Usually, the default for many OpenMP implementations is to parallelize the above iterations as block of consecutive n/thread_count iterations to each thread.
- What if f(i) has latency that increases with i? What is the best schedule then?

Example of function *f*.

```
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 */
```

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

Wouldn't this be better? (why?)

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.

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

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);
```

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.

The Static Schedule Type

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

`schedule(static,1)`

Thread 0 : 0, 3, 6, 9

Thread 1 : 1, 4, 7, 10

Thread 2 : 2, 5, 8, 11

`schedule(static,2)`

Thread 0 : 0, 1, 6, 7

Thread 1 : 2, 3, 8, 9

Thread 2 : 4, 5, 10, 11

`schedule(static,4)`

Thread 0 : 0, 1, 2, 3

Thread 1 : 4, 5, 6, 7

Thread 2 : 8, 9, 10, 11

How to implement the default block scheduling using the static schedule?

The Dynamic Schedule Type

- The iterations are 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 runtime system.
- This continues until all the iterations are completed.
- The chunksize can be omitted. When it is omitted, a default 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.
- 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.

Example:

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

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

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.
- Example:

  export OMP_SCHEDULE ="static,1"

Another Way for Controlling the Schedule

- Using the omp_set_schedule function. Syntax:

```
void omp_set_schedule(omp_sched_t kind,
                      int chunk_size);
```

- Where `kind` is one of:
  - `omp_sched_static`
  - `omp_sched_dynamic`
  - `omp_sched_guided`
  - `omp_sched_auto`

Keep in mind:

- There is an overhead in using the schedule directive
- The overhead is higher in dynamic than static schedules
- The overhead of guided is the greatest of all three.
- So: if we get satisfactory performance without schedule then don't use schedule.

Rules of thumb

- If each iteration requires roughly the same amount of computation → default is best
- If the cost of each iteration increases/decreases linearly as the loop executes → static with small chunksize
- If the cost cannot be determined → you need to try several schedules: schedule(runtime) and try different options with OMP_SCHEDULE

Question

Can we parallelize the following loop? If yes, do it. If not, why not?

```
a[0] = 0;
for( i = 1; i < n; i++)
    a[i] = a[i-1] + i;
```

Hint:

$a[1] = a[0] + 1$

$a[2] = a[1] + 2 = a[0] + 1 + 2$

$a[3] = a[2] + 3 = a[0] + 1 + 2 + 3$

$a[i] = 0 + 1 + 2 + \dots + i = i(i+1)/2$

Conclusions

- OpenMP depends on compiler directives, runtime library, and environment variable.
- The main concept to parallelize a program with OpenMP is how to have independent for-loops.
- May aspects of OpenMP are still implementation dependent, so you need to be careful!