Shared Memory Programming with OpenMP

Moreno Marzolla
Dip. di Informatica—Scienza e Ingegneria (DISI)
Università di Bologna

moreno.marzolla@unibo.it
Credits

- Peter Pacheco, Dept. of Computer Science, University of San Francisco
  http://www.cs.usfca.edu/~peter/
- Mary Hall, School of Computing, University of Utah
  https://www.cs.utah.edu/~mhall/
- Salvatore Orlando, DAIS, Università Ca' Foscari di Venezia, http://www.dais.unive.it/~calpar/
- Tim Mattson, Intel
- Blaise Barney, OpenMP
  https://computing.llnl.gov/tutorials/openMP/ (highly recommended!!)
OpenMP

- Model for shared-memory parallel programming
- Portable across shared-memory architectures
- Incremental parallelization
  - Parallelize individual computations in a program while leaving the rest of the program sequential
- Compiler based
  - Compiler generates thread programs and synchronization
- Extensions to existing programming languages (Fortran, C and C++)
  - mainly by directives (#pragma omp ...)
  - a few library routines
OpenMP Timeline

• Initially, the API specifications were released separately for C and Fortran

• Since 2005, they have been released together
  - Oct 1997    Fortran 1.0
  - Oct 1998    C/C++ 1.0
  - Nov 1999    Fortran 1.1
  - Nov 2000    Fortran 2.0
  - Mar 2002    C/C++ 2.0
  - May 2005    OpenMP 2.5 (GCC $\geq$ 4.2.0)
  - May 2008    OpenMP 3.0 (GCC $\geq$ 4.4.0)
  - Jul 2011    OpenMP 3.1 (GCC $\geq$ 4.7.0)
  - Jul 2013    OpenMP 4.0 (GCC $\geq$ 4.9.1)
  - Nov 2015    OpenMP 4.5 (GCC $\geq$ 6.1)
Most OpenMP programs only use these items

<table>
<thead>
<tr>
<th>OpenMP pragma, function, or clause</th>
<th>Concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma omp parallel</td>
<td>Parallel region, teams of threads, structured block, interleaved execution across threads</td>
</tr>
<tr>
<td>int omp_get_thread_num()</td>
<td>Create threads with a parallel region and split up the work using the number of threads and thread ID</td>
</tr>
<tr>
<td>int omp_get_num_threads()</td>
<td>Timing blocks of code</td>
</tr>
<tr>
<td>double omp_get_wtime()</td>
<td>Set the default number of threads with an environment variable</td>
</tr>
<tr>
<td>setenv OMP_NUM_THREADS=N</td>
<td>Synchronization, critical sections</td>
</tr>
<tr>
<td>export OMP_NUM_THREADS=N</td>
<td>Worksharing, parallel loops</td>
</tr>
<tr>
<td>reduction(op:list)</td>
<td>Reductions of values across a team of threads</td>
</tr>
<tr>
<td>schedule(dynamic [,chunk])</td>
<td>Loop schedules</td>
</tr>
<tr>
<td>schedule(static [,chunk])</td>
<td>Data environment</td>
</tr>
<tr>
<td>private(list), shared(list),</td>
<td></td>
</tr>
<tr>
<td>firstprivate(list)</td>
<td>Worksharing with a single thread</td>
</tr>
<tr>
<td>#pragma omp master</td>
<td>Tasks including the data environment for tasks.</td>
</tr>
<tr>
<td>#pragma omp single</td>
<td></td>
</tr>
<tr>
<td>#pragma omp task</td>
<td></td>
</tr>
<tr>
<td>#pragma omp taskwait</td>
<td></td>
</tr>
</tbody>
</table>

Credits: Tim Mattson
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
  – Provide synchronization constructs

• OpenMP will not:
  – Parallelize automatically
  – Guarantee speedup
  – Avoid data races
OpenMP Execution Model

- **Fork-join model** of parallel execution
- Begin execution as a single process (master thread)
- Start of a parallel construct:
  - Master thread creates team of threads (worker threads)
- Completion of a parallel construct:
  - Threads in the team synchronize – **implicit barrier**
- Only the master thread continues execution
OpenMP uses Pragmas

```
#pragma omp construct [clause [clause ...]]
```

- **Pragmas** are special preprocessor instructions
  - They allow behaviors that are not part of the C specification
- Compilers that don’t support the *pragmas* ignore them
- Most OpenMP constructs apply to the *structured block* following the directive
  - Structured block: a block of one or more statements with **one point of entry** at the top and **one point of exit** at the bottom
  - Returning from inside a parallel block is not allowed
The \texttt{#pragma omp parallel} directive

- When a thread reaches a \texttt{parallel} directive, it creates a pool of threads and becomes the master of the team
  - The master has thread ID 0
- The default pool size is implementation-dependent
- The code of the parallel region is duplicated and all threads will execute it
- There is an implied barrier at the end of a parallel section. Only the master continues execution past this point

\begin{verbatim}
#pragma omp parallel [clause ...]
clause ::= if (scalar_expression) |
          private (list) |
          shared (list) |
          default (shared | none) |
          firstprivate (list) |
          reduction (operator: list) |
          copyin (list) |
          num_threads(thr)
\end{verbatim}
"Hello, world" in OpenMP

```c
/* omp-demo0.c */
#include <stdio.h>

int main( void )
{
    #pragma omp parallel
    {
        printf("Hello, world!\n");
    }

    return 0;
}
```

$ gcc -fopenmp omp-demo0.c -o omp-demo0
$ ./omp-demo0
Hello, world!
Hello, world!

$ OMP_NUM_THREADS=4 ./omp-demo0
Hello, world!
Hello, world!
Hello, world!
Hello, world!
“Hello, world” in OpenMP

```c
/* omp-demo1.c */
#include <stdio.h>
#include <omp.h>

void say_hello( void )
{
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();
    printf("Hello from thread %d of %d\n", my_rank, thread_count);
}

int main( void )
{
    #pragma omp parallel
    say_hello();

    return 0;
}
```

$ gcc -fopenmp omp-demo1.c -o omp-demo1$
$ ./omp-demo1
Hello from thread 0 of 2
Hello from thread 1 of 2

$ OMP_NUM_THREADS=4 ./omp-demo1
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4
/* omp-demo2.c */
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void say_hello( void )
{
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();
    printf("Hello from thread %d of %d\n", my_rank, thread_count);
}

int main( int argc, char* argv[] )
{
    int thr = atoi( argv[1] );
    #pragma omp parallel num_threads(thr)
    say_hello();
    return 0;
}
Setting the number of threads programmatically

```c
/* omp-demo3.c */
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void say_hello( void )
{
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();
    printf("Hello from thread %d of %d\n",
            my_rank, thread_count);
}

int main( int argc, char* argv[] )
{
    omp_set_num_threads(4);
    #pragma omp parallel
    say_hello();

    return 0;
}
```

$ gcc -fopenmp omp-demo3.c -o omp-demo3
$ OMP_NUM_THREADS=8 ./omp-demo3
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4
Warning

- `omp_get_num_threads()` returns the number of threads in the currently active pool
  - If no thread pool is active, the function returns 1
- `omp_get_max_threads()` returns the maximum number of threads that can be created

```c
int main( int argc, char* argv[] )
{
    printf("Before par region: threads=%d, max=%d\n", omp_get_num_threads(), omp_get_max_threads());
    #pragma omp parallel
    {
        printf("Inside par region: threads=%d, max=%d\n", omp_get_num_threads(), omp_get_max_threads());
    }
    return 0;
}
```
See `omp-demo4.c`
Nested parallelism

- It is possible to nest parallel regions
  - Nesting must be enabled by setting the environment variable `OMP_NESTED=true`
  - `omp_get_num_threads()` returns the number of the innermost thread pool a thread is part of
  - `omp_get_thread_num()` returns the ID of a thread in the innermost pool it is part of
- Nested parallelism could cause oversubscription
  - more running threads than processor cores
- See `omp-nested.c`
More complex example

```c
int num_thr = 3
#pragma omp parallel if(num_thr>=4) num_threads(num_thr)
{
    /* parallel block */
}
```

• The “if” clause is evaluated
  - If the clause evaluates to *true*, the `parallel` construct is enabled with `num_thr` threads
  - If the clause evaluates to *false*, the `parallel` construct is ignored
Taking times

double tstart, tstop;
tstart = omp_get_wtime();
#pragma omp ...
{
    /* block of code to measure */
}
/* block of code to measure */
tstop = omp_get_wtime();
printf("Elapsed time: %f\n", tstop - tstart);

Remember: implicit barrier here
Checking compiler support for OpenMP

```c
#ifdef __OPENMP
#include <omp.h>
#endif

/* ... */

#ifdef __OPENMP
    int my_rank =omp_get_thread_num();
    int thread_count =omp_get_num_threads();
#else
    int my_rank = 0;
    int thread_count = 1;
#endif
```
Scoping of variables
Scope

• In serial programming, the scope of a variable consists of the parts of a program where the variable can be used

• In OpenMP, the scope of a variable refers to the set of threads that can access the variable

• By default:
  - All variables that are visible at the beginning of a parallel block are shared across threads
  - All variables defined inside a parallel block are private to each thread
Variables scope in OpenMP

- **shared**(x) Default
  - all threads access the same memory location

- **private**(x)
  - each thread has its own private copy of x
  - all local instances of x are not initialized
  - local updates to x are lost when exiting the parallel region
  - the original value of x is retained at the end of the block (OpenMP ≥ 3.0 only)

- **firstprivate**(x)
  - each thread has its own private copy of x
  - all copies of x are initialized with the current value of x
  - local updates to x are lost when exiting the parallel region
  - the original value of x is retained at the end of the block (OpenMP ≥ 3.0 only)

- **default**(shared) or **default**(none)
  - affects all the variables not specified in other clauses
  - **default**(none) ensures that you must specify the scope of each variable used in the parallel block that the compiler can not figure out by itself (highly recommended!!)
What is the output of this program?

```c
/* omp-scope.c */
#include <stdio.h>

int main( void )
{
    int a=1, b=1, c=1, d=1;
    #pragma omp parallel num_threads(10) \ 
    private(a) shared(b) firstprivate(c)
    {
        printf("Hello World!\n");
        a++;
        b++;
        c++;
        d++;
    }
    printf("a=%d\n", a);
    printf("b=%d\n", b);
    printf("c=%d\n", c);
    printf("d=%d\n", d);
    return 0;
}
```

Hint: compile with -Wall
GCC annoys us

- When using `default(none)`
  - In GCC < 9.x, `const` variables are predetermined shared
  - In GCC ≥ 9.x, you must explicitly specify the visibility of every variable
- See: https://www.gnu.org/software/gcc/gcc-9/porting_to.html
GCC annoys us

- GCC < 9.x: OK
- GCC ≥ 9.x:
  error: ‘foo’ not specified in enclosing ‘parallel’

```c
const int foo = 1;
#pragma omp parallel default(none)
{
    int baz = 0;
    baz += foo;
}
```

See `omp-bug1.c`
GCC annoys us

- GCC < 9.x: ‘foo’ is predetermined ‘shared’ for ‘shared’
- GCC $\geq$ 9.x: OK

```
const int foo = 1;
#pragma omp parallel default(none) shared(foo)
{
    int baz = 0;
    baz += foo;
}
```

See omp-bug2.c
GCC annoys us

- GCC < 9.x: OK
- GCC ≥ 9.x: OK

```c
const int foo = 1;
#pragma omp parallel default(none) firstprivate(foo)
{
    int baz = 0;
    baz += foo;
}
```

See `omp-bug3.c`
Example: the trapezoid rule
The trapezoid rule

/* Serial trapezoid rule */
h = (b-a)/n;
result = 0;
x = a;
for (i=0; i<n; i++) {
    result += h*(f(x) + f(x+h))/2;
    x += h;
}
return result;

See trap.c
A first OpenMP version

- Two types of tasks
  - computation of the areas of individual trapezoids
  - adding the areas of trapezoids
- Areas can be computed independently
  - *embarrassingly parallel* problem
- We assume that there are more trapezoids than OpenMP threads ($n \gg P$)

```c
/* Serial trapezoid rule */
h = (b-a)/n;
result = 0;
x = a;
for (i=0; i<n; i++) {
    result += h*(f(a) + f(a+h))/2;
    x += h;
}
return result;
```
Assigning trapezoids to threads
A first OpenMP version

- Split the $n$ intervals across OpenMP threads
- Thread $t$ stores its result in `partial_result[t]`
- The master sums all partial results
- See `omp-trap0.c`
  - Try adding "default(none)" to the `omp parallel` clause
A second OpenMP version

- Split the $n$ intervals across OpenMP threads
- Thread $t$...
  - ...stores its result in a local variable `partial_result`
  - ...updates the global result
- **omp atomic**
  - Protects updates to a shared variable
  - Updates must be of the form "read-update-write", e.g., `var += x`
- **omp critical**
  - Protects access to a critical section, which may consist of arbitrary instructions
  - All threads will eventually execute the critical section; however, only one thread at a time can be inside the critical block
- **critical** protects code; **atomic** protects memory locations
- See `omp-trap1.c`
The **atomic** directive

- The **omp atomic** directive ensures that only one thread at the time updates a shared variable

```c
#pragma omp parallel
{
    double partial_result = trap(a, b, n);
    #pragma omp atomic
    result += partial_result;
}
```

- The code above forces all threads to serialize during the update of the shared variable
  - This is not a real problem, since each thread will update the shared variable exactly once
- We can also use the **reduction** clause
The **reduction** clause

- `reduction( <op> : <variable> )`

  can be one of `+`, `*`, `|`, `^`, `&`, `|`, `&&`, `||`, (in principle also subtraction, but the OpenMP specification does not guarantee the result to be uniquely determined)

```c
#pragma omp parallel reduction(+:result)
{
  double partial_result = trap(a, b, n);
  result += partial_result;
}
```

See `omp-trap2.c`
Reduction operators

- A reduction operator is a binary associative operator such as addition or multiplication
  - An operator $\diamond$ is associative if $(a \diamond b) \diamond c = a \diamond (b \diamond c)$
- A reduction is a computation that repeatedly applies the same reduction operator to a sequence of operands to get a single result
  - $\diamond$-reduce$(x_0, x_1, \ldots, x_{n-1}) = x_0 \diamond x_1 \diamond \ldots \diamond x_{n-1}$
How the **reduction** clause works

- One *private* copy of the reduction variable is created for each thread.
- Each private copy is initialized with the neutral element of the reduction operator (e.g., 1 for *, 0 for +)
- Each thread executes the parallel region.
- When all threads finish, the reduction operator is applied to the *last* value of each local reduction variable, *and* the value the reduction variable had *before* the parallel region.

```c
/* omp-reduction.c */
#include <stdio.h>
int main( void )
{
    int a = 2;
    #pragma omp parallel reduction(*:a)
    {
        a += 2;
    }
    printf("%d\n", a);
    return 0;
}
```

$ OMP_NUM_THREADS=1 ./omp-reduction 6 $ OMP_NUM_THREADS=2 ./omp-reduction 18 $ OMP_NUM_THREADS=4 ./omp-reduction 162
How the **reduction** clause works

```c
/* omp-reduction.c */
#include <stdio.h>
int main( void )
{
    int a = 2;
    #pragma omp parallel reduction(*:a)
        {
            /* implicit initialization a = 1 */
            a += 2;
        }
    printf("%d\n",a);
    return 0;
}
```

### Diagram:

```
OMP_NUM_THREADS=2

```

```
\[
\begin{array}{ccc}
\text{a = 2} & \text{a = 1} & \text{a = 1} \\
\downarrow & \downarrow & \downarrow \\
\text{a = 1} & \text{a += 2} & \text{a += 2} \\
\downarrow & \downarrow & \downarrow \\
\text{a = 2 * 3 * 3} & \text{Implicit initialization} & \\
\end{array}
\]
```
Some valid reduction operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>min</td>
<td>largest positive number</td>
</tr>
<tr>
<td>max</td>
<td>most negative number</td>
</tr>
<tr>
<td>&amp;</td>
<td>~0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>0</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The **omp for** directive

- The **omp for** directive is used inside a parallel block
- Loop iterations are assigned to the threads of the current team (the ones created with **omp parallel**)
  - The loop variable is made **private** by default

```c
#pragma omp parallel
{
#pragma omp for
  for ( i=0; i<n; i++) {
    do_work(i);
  }
} // Implied barriers here
```

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>do_work(0)</td>
<td>do_work(4)</td>
<td>do_work(8)</td>
</tr>
<tr>
<td>do_work(1)</td>
<td>do_work(5)</td>
<td>do_work(9)</td>
</tr>
<tr>
<td>do_work(2)</td>
<td>do_work(6)</td>
<td>do_work(10)</td>
</tr>
<tr>
<td>do_work(3)</td>
<td>do_work(7)</td>
<td>do_work(11)</td>
</tr>
</tbody>
</table>

OpenMP Programming 40
The parallel for directive

- The parallel and for directives can be collapsed in a single parallel for

```c
double trap( double a, double b, int n )
{
    double result = 0;
    const double h = (b-a)/n;
    int i;
    #pragma omp parallel for reduction(+:result)
    for ( i = 0; i<n-1; i++ ) {
        result += h*(f(a+i*h) + f(a+(i+1)*h))/2;
    }
    return result;
}
```

See omp-trap3.c
Legal forms for parallelizable *for* statements

- **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, *incr* must have integer type
- The expressions *start*, *end*, and *incr* must not change during execution of the loop
- Variable `index` can only be modified by the “increment expression” in the “for” statement

\[
\begin{align*}
\text{for} & \quad \text{index} = \text{start} \; ; \\
& \quad \text{index} < \text{end} \\
& \quad \text{index} \leq \text{end} \\
& \quad \text{index} > \text{end} \\
& \quad \text{index} \geq \text{end}
\end{align*}
\]

- `index++`
- `++index`
- `index--`
- `--index`
- `index += incr`
- `index -= incr`
- `index = index + incr`
- `index = incr + index`
- `index = index - incr`
Data dependencies

- It is not possible to use a `parallel for` directive if data dependencies are present.
- Example: computation of PI

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

```c
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;
```
Removing the data dependency

```c
double factor;
double sum = 0.0;
#pragma omp parallel for private(factor) reduction(+:sum)
for (k=0; k<n; k++) {
    if ( k % 2 == 0 ) {
        factor = 1.0;
    } else {
        factor = -1.0;
    }
    sum += factor/(2*k + 1);
}
pi_approx = 4.0 * sum;
```

*factor must have private scope*
Can my loop be parallelized?

• We will devote a whole lecture to this problem
  – Stay tuned...
• A quick-and-dirty test: run the loop backwards
  – If the program is still correct, the loop might be parallelizable
  – Not 100% reliable, but works most of the time

```c
for (i=0; i<n; i++) {
    /* loop body */
}
```

```
for (i=n-1; i>=0; i--) {
    /* loop body */
}
```
Scheduling loops

```c
/* omp-mandelbrot.c */
...

int main( int argc, char *argv[] )
{
    int x, y;

    gfx_open( xsize, ysize, "Mandelbrot Set");
    #pragma omp parallel for private(x) schedule(dynamic, 64)
    for ( y = 0; y < ysize; y++ ) {
        for ( x = 0; x < xsize; x++ ) {
            drawpixel( x, y );
        }
    }
    printf("Click to finish\n");
    gfx_wait();
    return 0;
}
```
**schedule**(*type*, *chunksize*)

- **type** can be:
  - *static*: the iterations can be assigned to the threads before the loop is executed. If *chunksize* is not specified, iterations are evenly divided contiguously among threads
  - *dynamic* or *guided*: iterations are assigned to threads while the loop is executing. Default *chunksize* is 1
  - *auto*: the compiler and/or the run-time system determines the schedule
  - *runtime*: the schedule is determined at run-time using the `OMP_SCHEDULE` environment variable (e.g., `export OMP_SCHEDULE="static,1"`)

- Default schedule type is implementation dependent
  - GCC seems to use *static* by default
Example

- Twelve iterations 0, 1, … 11 and three threads
- `schedule(static, 1)`

```
0  1  2  3  4  5  6  7  8  9  10  11
```

- `schedule(static, 2)`

```
0  1  2  3  4  5  6  7  8  9  10  11
```

- `schedule(static, 4)`

```
0  1  2  3  4  5  6  7  8  9  10  11
```

Default chunksize in this case
The Dynamic/Guided Schedule Types

• The iterations are broken up into chunks of `chunksize` consecutive iterations
  – However, in a `guided` schedule, as chunks are completed the size of the new chunks decreases

• Each thread executes a chunk, and when a thread finishes a chunk, it requests another one from the runtime system
  – Master/Worker paradigm
Choosing a schedule clause

<table>
<thead>
<tr>
<th>Schedule clause</th>
<th>When to use</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td>Pre-determined and predictable work per iteration</td>
<td>Least work at runtime: scheduling done at compile-time</td>
</tr>
<tr>
<td>dynamic</td>
<td>Unpredictable, highly variable work per iteration</td>
<td>Most work at runtime: complex scheduling logic used at run-time</td>
</tr>
</tbody>
</table>

Credits: Tim Mattson
Choosing a block size

- Too small = high scheduling overhead
- Too large = unbalanced workload

"Optimal" block size

Diagram showing the relationship between block size and wall-clock time, with an optimal block size indicated where wall-clock time is minimized.
The **collapse** directive

- Specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause

```c
#pragma omp parallel for collapse(2)
for ( y = 0; y < ysize; y++ ) {
    for ( x = 0; x < xsize; x++ ) {
        drawpixel( x, y );
    }
}
```

collapse(2) makes x and y private by default
How **collapse** works

```c
#pragma omp parallel for collapse(2)
for (i=0; i<4; i++) {
    for (j=0; j<5; j++) {
        do_work(i,j);
    }
}
```

See [omp-mandelbrot.c](omp-mandelbrot.c)
Spot the bug

```c
#pragma omp parallel for private(temp)
for (i=0; i<n; i++){
    for (j=0; j<m; j++){
        temp = b[i]*c[j];
        a[i][j] = temp * temp + d[i];
    }
}
```
Example: Odd-Even Tranposition Sort
Serial Odd-Even Transposition Sort

- Variant of bubble sort
- Compare all (even, odd) pairs of adjacent elements, and exchange them if in the wrong order
- Then compare all (odd, even) pairs, exchanging if necessary; repeat the step above

```
```

Time

Compare and exchange
Serial Odd-Even Transposition Sort

- Variant of bubble sort
- Compare all (even, odd) pairs of adjacent elements, and exchange them if in the wrong order
- Then compare all (odd, even) pairs, exchanging if necessary; repeat the step above

```c
for (phase = 0; phase < n; phase++) {
    if ( phase % 2 == 0 ) {
        for (i=0; i<n-1; i+=2) {
            if (v[i] > v[i+1]) swap( &v[i], &v[i+1] );
        }
    } else {
        for (i=1; i<n-1; i+=2) {
            if (v[i] > v[i+1]) swap( &v[i], &v[i+1] );
        }
    }
}
```
First OpenMP Odd-Even Sort

```c
for (phase = 0; phase < n; phase++) {
    if ( phase % 2 == 0 ) {
        #pragma omp parallel for default(none) shared(v,n)
        for (i=0; i<n-1; i+=2) {
            if (v[i] > v[i+1]) swap( &v[i], &v[i+1] );
        }
    } else {
        #pragma omp parallel for default(none) shared(v,n)
        for (i=1; i<n-1; i+=2) {
            if (v[i] > v[i+1]) swap( &v[i], &v[i+1] );
        }
    }
}
```
for (phase = 0; phase < n; phase++) {
    if (phase % 2 == 0) {
        #pragma omp parallel for default(none) shared(v,n,phase) private(i)
        for (i=0; i<n-1; i+=2) {
            if (v[i] > v[i+1]) swap( &v[i], &v[i+1] );
        }
    } else {
        #pragma omp parallel for default(none) shared(v,n,phase) private(i)
        for (i=1; i<n-1; i+=2) {
            if (v[i] > v[i+1]) swap( &v[i], &v[i+1] );
        }
    }
}
First OpenMP Odd-Even Sort

- The pool of threads is being created/destroyed at each `omp parallel for` region.
- This *may* produce some overhead, depending on the OpenMP implementation.
- You can nest `omp for` inside `omp parallel` to recycle the threads from the same pool.
Second OpenMP Odd-Even Sort

```c
#pragma omp parallel default(none) shared(v,n) private(phase)
for (phase = 0; phase < n; phase++) {
    if ( phase % 2 == 0 ) {
        #pragma omp for
        for (i=0; i<n-1; i+=2) {
            if (v[i] > v[i+1]) swap( &v[i], &v[i+1] );
        }
    } else {
        #pragma omp for
        for (i=1; i<n-1; i+=2) {
            if (v[i] > v[i+1]) swap( &v[i], &v[i+1] );
        }
    }
}
```

See `omp-odd-even.c`
#pragma omp parallel num_threads(4) \
    default(none) private(i,j)
for (i = 0; i < n; i++) {
    #pragma omp for
    for (j=0; j < 100; j++) {
        do_work(i, j);
    }
}
Work-sharing constructs
The `sections` directive

- Specifies that the enclosed section(s) of code are to be divided among the threads in the team
- Independent `section` directives are nested within a `sections` directive
  - Each `section` is executed once by one thread
  - Different `section` may be executed by different threads
  - A thread might execute more than one `section` if it is quick enough and the implementation allows that
- There is an implied barrier at the end of a `sections` directive, unless the `nowait` clause is used
Example

- The execution order of tasks is *non deterministic*
- Assignment of tasks to threads is *non deterministic*

```c
#pragma omp parallel num_threads(2)
{
    #pragma omp sections
    {
        #pragma omp section
        fred();
        #pragma omp section
        barney();
        #pragma omp section
        wilma();
        #pragma omp section
        betty();
    }
}
```

Implicit barrier here: fred(), barney(), wilma() and betty() must all complete before leaving this point.
# Example

```
#define N 1000
int main(void)
{
    int i;
    float a[N], b[N], c[N], d[N];
    /* Some initializations */
    for (i=0; i < N; i++) {
        a[i] = i * 1.5;
        b[i] = i + 22.35;
    }
    #pragma omp parallel private(i)
    {
        #pragma omp sections nowait
        {
            #pragma omp section
            for (i=0; i < N; i++)
                c[i] = a[i] + b[i];
            #pragma omp section
            for (i=0; i < N; i++)
                d[i] = a[i] * b[i];
        } /* end of sections */
    } /* end of parallel section */
    return 0;
}
```
OpenMP synchronization

- **#pragma omp barrier**
  - All threads in the currently active team must reach this point before they are allowed to proceed

- **#pragma omp master**
  - Marks a parallel region which is executed by the master only (the thread with rank = 0); other threads just skip the region
  - There is no implicit barrier at the end of the block

- **#pragma omp single**
  - Marks a parallel region which is executed once by the first thread reaching it, whichever it is
  - A barrier is implied at the end of the block
```c
#pragma omp parallel
{
    compute();
    #pragma omp master
    swap();
    #pragma omp barrier
    more_compute();
}
```
Example

- The code from `omp-trap0.c` can be rewritten as follows
  - probably less clear and not as efficient as the original version

```c
#pragma omp parallel
{
    int thr = omp_get_thread_num();
    partial_result[thr] = trap(a, b, n);
    #pragma omp barrier
    #pragma omp master
    {
        result = 0.0;
        for (i=0; i<thread_count; i++)
            result += partial_result[i];
    }
}
```
OpenMP tasking constructs

• Not all programs have simple loops that OpenMP can parallelize

• Example: linked list traversal
  – Each node of the linked list is "processed" independently from other nodes

```c
p = head;
while (p) {
    processwork(p);
    p = p->next;
}
```

• OpenMP `parallel for` works only for loops where the iteration count can be known in advance at runtime

Credits: Tim Mattson
What are OpenMP tasks?

- Tasks are independent units of work
- Tasks are composed of:
  - code to execute
  - data to compute with
- Threads are assigned to perform the work of each task
  - The thread that encounters the task construct may execute the task immediately
  - The threads may defer execution until later
- Tasks can be nested: a task may generate other tasks

Credits: Tim Mattson
#pragma omp task

```
#pragma omp parallel
{
    #pragma omp master
    {
        #pragma omp task
        fred();
        #pragma omp task
        barney();
        #pragma omp task
        wilma();
    }
}
```

- Create a pool of threads
- Only the master forks threads
- Tasks executed by some threads in some order
- All tasks complete before this barrier is released
Differences between tasks and sections

- OpenMP sections must be completed (unless the `nowait` clause is used) inside the `sections` block.
- OpenMP tasks are queued and completed whenever possible.
  - The OpenMP runtime may choose to move tasks between threads even during execution.
Data scoping with tasks

• Variables can be shared, private or firstprivate with respect to task
  – If a variable is shared on a task construct, the references to it inside the construct are to the storage with that name at the point where the task was encountered
  – If a variable is private on a task construct, the references to it inside the construct are to new uninitialized storage that is created when the task is executed
  – If a variable is firstprivate on a construct, the references to it inside the construct are to new storage that is created and initialized with the value of the existing storage of that name when the task is encountered
Data scoping with tasks

- The behavior you want for tasks is usually `firstprivate`, because the task may not be executed until later (and variables may have gone out of scope)
  - Variables that are `private` when the task construct is encountered are `firstprivate` by default

```c
#pragma omp parallel shared(A) private(B)
{
  ...
  #pragma omp task
  {
    int C;
    compute(A, B, C);
  }
}
```

- `A` is shared
- `B` is `firstprivate`
- `C` is `private`
Linked list traversal with tasks

```
#pragma omp parallel
{
  #pragma omp single
  {
    p=head;
    while (p) {
      #pragma omp task firstprivate(p)
      processwork(p);
      p = p->next;
    }
  }
}
```

Creates a task with its own copy of “p” initialized to the value of “p” when the task is defined.
When/Where are tasks completed?

- At thread barriers (explicit or implicit)
  - e.g., at the end of a `#pragma omp parallel` block
  - applies to all tasks generated in the current parallel region up to the barrier
- At `taskwait` directive
  - wait until all tasks defined in the current task (not in the descendants!) have completed.
  - `#pragma omp taskwait`
  - The code executed by a thread in a parallel region is considered a task here

Credits: Tim Mattson
Example

```c
#pragma omp parallel
{
    #pragma omp master
    {
        #pragma omp task
        fred();
        #pragma omp task
        barney();
        #pragma omp taskwait
    }
    #pragma omp task
    wilma();
}
```

Fred() and Barney() must complete before Wilma() starts
Example: parallel Fibonacci with tasks

\[ F_n = F_{n-1} + F_{n-2} \]
\[ F_0 = F_1 = 1 \]

Inefficient algorithm \(O(2^n)\)

```c
#include <stdio.h>

int fib(int n)
{
    int n1, n2;
    if (n < 2) {
        return 1;
    } else {
        n1 = fib(n-1);
        n2 = fib(n-2);
        return n1 + n2;
    }
}

int main(int argc, char* argv[])
{
    int n = 10, res;
    res = fib(n);
    printf("fib(%d)=%d\n", n, res);
    return 0;
}
```

Credits: Tim Mattson
Example: parallel Fibonacci with tasks

- Binary tree of tasks
- A task cannot complete until all tasks below it in the tree are complete
  - enforced with `taskwait`
- `n1`, `n2` are local, and so by default they are private to current task
  - must be `shared` on child tasks so they don’t create their own `firstprivate` copies

```c
#include <stdio.h>

int fib( int n )
{
    int n1, n2;
    if (n < 2) {
        return 1;
    } else {
        #pragma omp task shared(n1)
        n1 = fib(n-1);
        #pragma omp task shared(n2)
        n2 = fib(n-2);
        #pragma omp taskwait
        return n1 + n2;
    }
}

int main( int argc, char* argv[] )
{
    int n = 10, res;
    #pragma omp parallel
    #pragma omp master
    res = fib(n);
    printf("fib(%d)=%d\n", n, res);
    return 0;
}
```

Credits: Tim Mattson
Concluding Remarks

• OpenMP is a standard for programming shared-memory systems.
  – Uses both special functions and preprocessor directives called pragmas.

• OpenMP programs start multiple threads
  – By default most systems use a block-partitioning of the iterations in a parallelized for loop
  – OpenMP offers a variety of scheduling options.

• In OpenMP the scope of a variable is the collection of threads to which the variable is accessible.

• A reduction is a computation that repeatedly applies the same reduction operator to a sequence of operands in order to get a single result.
References

- An excellent tutorial on OpenMP is available at: https://computing.llnl.gov/tutorials/openMP/