High-Performance Computing

– Shared Memory Programming with OpenMP –

Christian Plessl

High-Performance IT Systems Group

Paderborn University

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

version 1.1.2 2017-11-27

Paderborn Center for Parallel Computing

Outline

- **Basic OpenMP (covered by Pacheco book)**
  - concepts
  - work sharing (loop parallelization)
  - variable scoping
- **More OpenMP (covered in future lecture)**
  - task parallelism
  - SIMD parallelism (vectorization)
  - task loops, do across loops
- **Advanced OpenMP (optionally covered)**
  - target offloading to GPUs

OpenMP

- **MP = multi-processing**
- **API for explicit multi-threaded, shared-memory parallel programming with three components**
  - compiler directives
  - runtime library functions
  - environment variables
- **Goals of OpenMP**
  - standardization and portability
    - jointly defined by a group of major hard ware and software vendors
    - widely supported on Unix/Linux and Windows
    - API available for C/C++ and Fortran
  - ease of use
    - a very small set of of directives is sufficient to cover many common cases
    - supports incremental parallelization
    - addresses coarse and fine-grained parallelism

OpenMP (2)

- **History**
  - in the early 1990's HPC vendors have developed different OpenMP-like compiler extensions for Fortran
  - mid 1990's begin of efforts for a common API for shared memory multi-threading
  - OpenMP 1.0 (1997/98) and OpenMP 2.0 (2000/2002) focus on parallelization of highly regular loops
  - OpenMP 3.0 (2008) introduces task-level parallelism
  - OpenMP 4.0 (2013) adds support target offloading for accelerators, SIMD (vectorization), user-defined reductions, …
  - OpenMP 4.5 (2015) introduces taskloops, do across loops, task priorities and improves target offloading

Hello world for OpenMP

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

void Hello(void);  /* Thread function */

int main(int argc, char* argv[]) {
   /* Get number of threads from command line */
   int thread_count = strtol(argv[1], NULL, 10);

#  pragma omp parallel num_threads(thread_count)
   Hello();

   return 0;
}  /* main */

void 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);

}  /* Hello */
```

OpenMP compiler directive

OpenMP runtime library functions

Compiling and Executing OpenMP Programs

```
gcc -g -Wall -fopenmp -o omp_hello omp_hello . c
```

compiling

```
./omp_hello 4
```

running with 4 threads

possible outcomes

```
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
```

```
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4
```

```
Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
```

OpenMP Compiler Directives (Pragmas)

- **OpenMP makes extensive use of compiler directives, e.g.**

```
#pragma omp parallel default(shared) private(a,b)
```

- **Compiler directives provide special instructions to the compiler that are not part of the C/C++ standard**
  - compilers that don't support the directives just ignore them
- **All OpenMP directives start with #pragma omp**
  - directives can be followed by further clauses to modify and customize the basic operation
- **Examples for purpose of compiler directives**
  - spawning a parallel region
  - dividing blocks of code among threads
  - distributing loop iterations between threads
  - serializing sections of code
  - synchronization of work among threads

OpenMP Runtime-Library Functions

- **Runtime-library functions allow OpenMP programs to query and configure the execution environment (OpenMP runtime system)**

```
#include <omp.h>
int omp_get_num_threads(void)
```

- **Examples for purpose of runtime-library functions**
  - setting and querying the number of threads
  - querying a thread's unique identifier (id), a thread's ancestor identifier, team size
  - querying if in a parallel region and at what level
  - setting and querying nested parallelism
  - setting, initializing and terminating locks
  - querying wall clock time and resolution

OpenMP Environment Variables

- **The OpenMP runtime system can be controlled by environment variables**

```
export OMP_NUM_THREADS=8
```

- **The properties affected by the environment variables can also be changed by runtime-library functions**
- **Examples for purpose of OpenMP environment variables**
  - setting the number of threads
  - specifying how loop iterations are divided
  - binding threads to processors and cores
  - enabling/disabling and controlling depth of nested parallelism
  - enabling/disabling dynamic threads
  - setting thread stack size
  - setting threads wait policy

The OpenMP Directive ”parallel”

- **# pragma omp parallel**
  - most basic parallelization directive
  - creates a number of threads that run the following structured block of code
  - the number of threads that are used is determined by the run-time system
- **Clauses are used to modify directives**
  - the num_threads clause can be (optionally) added to a parallel directive
  - specifies number of threads that should execute the following block

```
# pragma omp parallel num_threads ( thread_count )
```

- **Notes**
  - the OpenMP standard doesn’t guarantee that this will actually start thread_count threads
  - that number of threads a program can start may be limited by the system
  - most current systems can start hundreds or even thousands of threads
  - unless we’re trying to start a lot of threads, we will almost always get the desired number of threads.

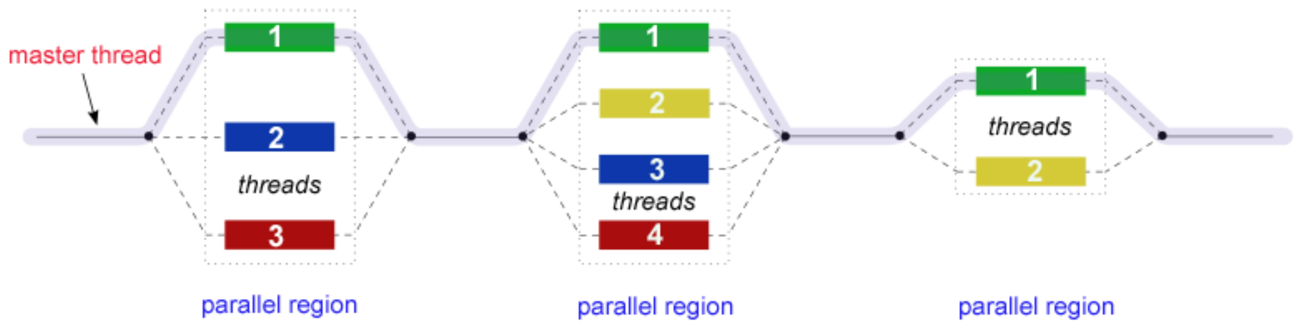
The OpenMP Directive ”parallel” (2)

- **For completeness: the complete specification of parallel directive is**

```
#pragma omp parallel [clause ...] newline
      if (scalar_expression)
      private (list)
      shared (list)
      default (shared | none)
      firstprivate (list)
      reduction (operator: list)
      copyin (list)
      num_threads (integer-expression)

structured_block
```

Fork-Join Model

- **The basic parallelization model in OpenMP is fork-join parallelism**
- **When master reaches the parallel directive:**
  - a collection of threads is created (denoted as team)
  - each child thread executes the code of the block that immediately follows the directive
  - the end of a parallel region is an implicit barrier, all threads are joined and the master thread continue



Fork-Join Model (2)

- **The actual number of threads in the team is determined by the following factors (in order of precedence)**
  - evaluation of the if clause
  - setting of the num_threads clause
  - use of the omp_set_num_threads() library function
  - setting of the OMP_NUM_THREADS environment variable
  - implementation on default or system configuration (typically number of cores)

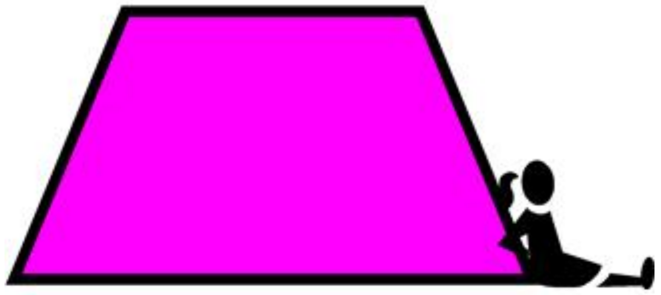
- **if clause**
  - the optional if clause can contain a boolean expression
  - a team is only created, if the clause evaluates to a non-zero value, otherwise the region is executed serially by the master thread

Writing Backward-Compatible Code

- **OpenMP is designed for backward compatibility, i.e. programs can be compiled with a compiler without OpenMP support**
    - #pragma omp directives are ignored
    - headers and library functions must be conditionally included
- **Conditional compilation**
    - compilers with OpenMP support define the symbol _OPENMP that can be used in the preprocessor

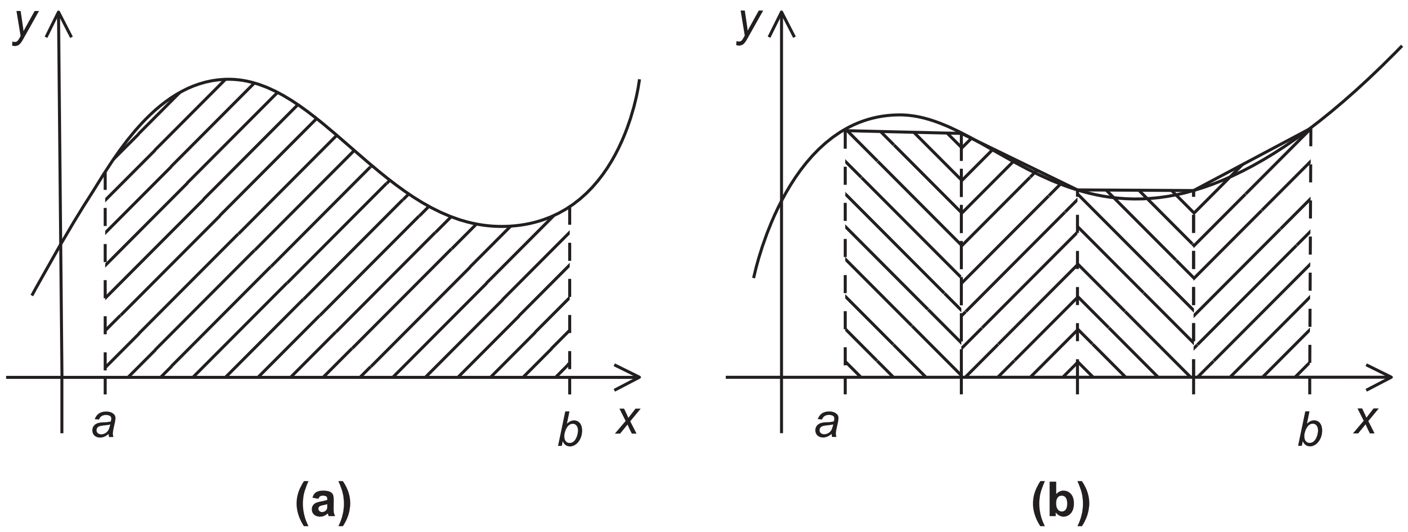
```
#ifdef _OPENMP
# include <omp.h>
#endif

# 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
```

The Trapezoidal Rule



The Trapezoidal Rule

- **Reminder: When discussing MPI we looked at numerical integration using the trapezoidal rule**



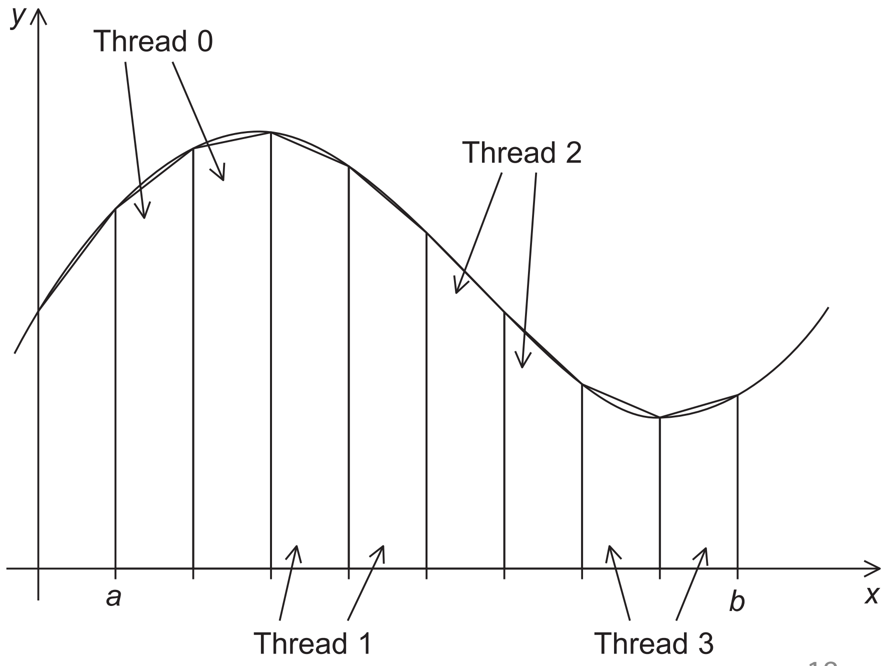
Sum of trapezoid areas $= h[f(x_0)/2 + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + f(x_n)/2]$.

Serial Algorithm

```
/* Input:  a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++) {
   x_i = a + i*h;
   approx += f(x_i);
}
approx = h*approx;
```

A First OpenMP Version

- **We identify two types of tasks:**
  1. computation of the areas of individual trapezoids, and
  2. adding the areas of trapezoids
- **There is no communication among the tasks computing the areas, but each of those tasks communicates with task to sum the area**
  - we assume that there are many more trapezoids than cores
  - hence, we aggregate tasks by assigning a contiguous block of trapezoids to each thread (and a single thread to each core)



A First OpenMP Version (2)

- When summing up the individual areas, we need to protect access to the shared sum variable to prevent a race condition

unpredictable results when two (or more) threads attempt to simultaneously execute:

```
global_result += my_result ;
```

- OpenMP provides the critical directive for protecting the block following the directive with a mutex
    - only one thread may enter the critical section at a time

```
# pragma omp critical
   global_result += my_result ;
```

A First OpenMP Version (3)

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

void Trap(double a, double b, int n, double* global_result_p);

int main(int argc, char* argv[]) {
    double  global_result = 0.0;
    double  a, b;
    int     n;
    int     thread_count;

    thread_count = strtol(argv[1], NULL, 10);
    printf("Enter a, b, and n\n");
    scanf("%lf %lf %d", &a, &b, &n);
#   pragma omp parallel num_threads(thread_count)
    Trap(a, b, n, &global_result);

    printf("With n = %d trapezoids, our estimate\n", n);
    printf("of the integral from %f to %f = %.14e\n",
       a, b, global_result);
    return 0;
}  /* main */
```

creates implicit tasks (later we will discuss how to create explicit tasks)

A First OpenMP Version (4)

```
void Trap(double a, double b, int n, double* global_result_p) {
   double  h, x, my_result;
   double  local_a, local_b;
   int  i, local_n;
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   h = (b−a)/n;
   local_n = n/thread_count;
   local_a = a + my_rank*local_n*h;
   local_b = local_a + local_n*h;
   my_result = (f(local_a) + f(local_b))/2.0;
   for (i = 1; i <= local_n−1; i++) {
     x = local_a + i*h;
     my_result += f(x);
   }
   my_result = my_result*h;

#  pragma omp critical
   *global_result_p += my_result;
}  /* Trap */
```

we created a multi-threaded version with only a couple of directives and minimal code changes

Scope of Variables in OpenMP

- **OpenMP variable scoping rules define how variables can be assigned by threads in a parallel block**
  - a variable that can be accessed by all the threads in the team has shared scope
  - a variable that can only be accessed by a single thread has private scope
  - variables declared
    - before a parallel block have a default scope of shared
    - within the block have default scope of private
- **Clauses in the OpenMP parallel directive can modify the scoping for variables**
  - private: new, uninitialized private variable of same type is created for each thread; the variable is created on the stack, i.e. not available when the thread enters the region the next time
  - shared: variable is shared among all treads in the team
  - default: allows to specify a default scope for all variables, "none" requires explicit scope decls.
  - firstprivate: like private, but with automatic initialization
  - lastprivate: like private but copies variable value at last loop iteration or section back to scope of main thread
  - copyin: for threadprivate variables (need to be declared before); works like firstprivate but the variable is allocated on the heap, i.e. the value persists between leaving and re-entering the parallel region

The Reduction Clause

- **We wanted to avoid the use of global variables in the Trapezoid Rule application**
  - hence we need to pass an additional shared pointer (global_result_p) to the Trap function
  - this pointer is used to update the global sum (protected by critical section)

```
void Trap(double a, double b, int n, double* global_result_p);
```

- **A more elegant solution would look like this**

```
double Trap(double a, double b, int n);
```

- **. . . and would be called like this**

```
global_result = Trap(a, b, n);
```

- **... but now we don‘t have a critical section anymore**

The Reduction Clause (2)

- If we use the following workaround, we force the threads to execute sequentially

```
    global_result = 0.0;
#   pragma omp parallel num_threads(thread_count)
    {
#      pragma omp critical
       global_result += Local_trap(double a, double b, int n);
    }
```

- We can avoid this problem by declaring a private variable inside the parallel block and moving the critical section after the function call

```
    global_result = 0.0;
#   pragma omp parallel num_threads(thread_count)
    {
       double my_result = 0.0;  /* private */
       my_result += Local_trap(double a, double b, int n);
#      pragma omp critical
       global_result += my_result;
    }
```

OpenMP Reductions

- **OpenMP reductions solve the problem in a more elegant and expressive way**
  - a reduction is a computation that repeatedly applies the same reduction operator to a sequence of operands in order to get a single result
  - the reduction operator is a binary operation (such as addition or multiplication)
  - the result of the reduction is stored in the reduction variable
- **To use a reduction, the parallel directive can be augmented with a reduction clause**

```
reduction(<operator>: <variable list>)
```

supported operators: +, *, -, &, |, ^, &&, ||

```
    global_result = 0.0;
#   pragma omp parallel num_threads(thread_count) \
       reduction(+: global_result)
    global_result += Local_trap(double a, double b, int n);
```

The "Parallel For" Directive

- **The #pragma omp parallel for directive forks a team of threads to execute the block**
  - the block must be a for loop
  - the directive 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;
```

$\longrightarrow$

```
   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;
```

parallel for directive applied to a for loop

Restrictions for Parallelizable For-Statements

- **The "Parallel for" directive works only for loops with simple control structure**
  - loop iteration variable must be an integer
  - the expressions start, end, and incr must not change during the loop and must have a compatible type
  - the loop variable index can only be modified by the "increment expression" in the for statement

```
     /                                              index++                \
    |                                               ++index                 |
    |                            index < end        index--                 |
    |                            index <= end       --index                 |
for |  index = start   ;         index >= end   ;   index += incr           |
    |                            index > end        index -= incr           |
    |                                               index = index + incr    |
    |                                               index = incr + index    |
     \                                              index = index - incr   /
```

- **Program correctness must not depend upon which thread executes a particular iteration**

Data dependencies

```
fib[0] = fib[1] = 1;
for (i=2; i<n; i++)
     fib[i] = fib[i-1] + fib[i-2];
```

$\downarrow$

note 2 threads

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

$\downarrow$

```
1 1 2 3 5 8 13 21 34 55
```

this is correct

$\downarrow$

but sometimes we get this

```
1 1 2 3 5 8 0 0 0 0
```

What happened?

- **OpenMP compilers don’t check for dependences among iterations in a loop that’s being parallelized with a parallel for directive**
  - dependencies in loops that cause the results of one or more loop iterations depend on other iterations are denoted as loop carried dependencies
  - in general, loops with loop carried dependencies cannot be correctly parallelized by OpenMP
- **Programmers need check dependencies of loops themselves**

Estimating π

- **Example from previous lectures**

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

```
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;
```

Can this loop be parallelized safely?

Or is there a loop carried dependency?

Estimating π: Solution #1

there is a loop carried 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;
```

Estimating π: Solution #2

- Remove loop carried dependency for factor
- Compute factor directly in each thread
- Potential for a difficult to find bug
  - by default all variables are shared among threads!
  - hence, the writes to factor in each thread are seen by the other threads in the team, which introduces data races
  - we need to explicitly declare factor as private to prevent this problem

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

ensures factor has private scope.

The default Clause

- Introducing data races by accident is a common problem
- One common technique to prevent this problem is using the OpenMP's **default clause** with option **none**

```
#omp parallel for … default (none)
```

- Explicitly choosing a scope of none (instead of relying on the default shared) requires the programmer to explicitly specify the scope of each variable in a block
    - this rule is enforced by the compiler

```
#omp parallel for default (none)
for(i=0; i<1024; i++) {
  a[i] = a[i] + 1;
}
```

error: no scope declared

```
#omp parallel for default (none)\
 private(i,a)
for(i=0; i<1024; i++) {
  a[i] = a[i] + 1;
}
```

Estimating π: Solution #3

```
      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);
      }
```

More About Loops in OpenMP: Sorting

- **Reminder: Odd-Even Transposition Sort (discussed in MPI lecture)**

```
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]);
```

	Subscript in Array						
Phase	**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

- **Does this algorithm have loop carried dependencies?**
  - outer loop has loop carried dependencies, result depends on execution order of iterations
  - inner loops does not have loop carried dependencies, all comparison and swaps can be execute in parallel or in arbitrary order → parallel for directive for inner loops should work fine
- **Potential problems**
  - all operations in inner loop must complete before next iteration of outer loop is started → guaranteed by implicit barrier after parallel for
  - overhead of spawning and joining threads in inner loop over and over again may be too high → we can keep the threads spawned, see Solution #2

OpenMP Odd-Even Sort: Solution #1

```
    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;
             }
          }
    }
```

spawning and joining threads of the inner loop in each phase again

OpenMP Odd-Even Sort: Solution #2

```
#  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;
            }
         }
   }
```

spawn threads once with "parallel" directive

reuse threads with "for" directive instead of "parallel for"

reuse threads with "for" directive instead of "parallel for"

Odd-Even Sort Performance Evaluation

- **Compare two solutions**
  - two "parallel for" directives (spawning and joining threads in each phase)
  - two "for" directives (reusing previously spawned threads)

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

time in seconds

- **Reusing threads shows significant performance benefits for this case study**

Scheduling Loops

- **Assume we want to parallelize the following loop with a (parallel) for directive**

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

- **Further assume that the time for evaluating f(i) increases linearly with the size of argument i**

```
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 */
```

Scheduling Loops (2)

- **The performance of the parallelized loop will depend strongly on the assignment**
  - block assignment leads to very imbalanced load, because thread 0 gets all the short function evaluations
  - cyclic assignment leads to a much more equally distributed load

thread	iterations (block assignment)	iterations (cyclic assignment)
0	0, 1, 2, 3, .. , (n/t)-1	0, n/t, 2n/t, …
1	n/t, (n/t)+1, (n/t)+2, …	1, (n/t)+1, (2n/t)+1, …
…		
t-1	(t-1)(n/t), (t-1)(n/t)+1, …, n-1	t-1, (n/t)+t, (2n/t)+t, …, n-1

OpenMP Schedule Clause

- **The OpenMP schedule clause allows the programmer to configure how loop iterations are assigned to threads**

```
schedule(<type> [, <chunksize>])
```

- **Type can be:**
  - static the iterations are assigned to the threads before the loop is executed (default)
  - 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**
  - only applicable to static, dynamic and guided

The Static Schedule Type

- **The static scheduler**
  - assigns chunks of chunksize iterations to each thread in round robin order
  - the assignment of chunks does not consider the actual workload load of the threads at runtime

- **Example: 12 iterations and 3 threads**

`schedule(static,1)`

$\text{Thread } 0: \quad 0, 3, 6, 9$

$\text{Thread } 1: \quad 1, 4, 7, 10$

$\text{Thread } 2: \quad 2, 5, 8, 11$

cyclic distribution

`schedule(static,2)`

$\text{Thread } 0: \quad 0, 1, 6, 7$

$\text{Thread } 1: \quad 2, 3, 8, 9$

$\text{Thread } 2: \quad 4, 5, 10, 11$

block-cyclic distribution

`schedule(static,4)`

$\text{Thread } 0: \quad 0, 1, 2, 3$

$\text{Thread } 1: \quad 4, 5, 6, 7$

$\text{Thread } 2: \quad 8, 9, 10, 11$

block distribution

The Dynamic Schedule Type

- **The iterations are also broken up into chunks of chunksize consecutive iterations**
  - each thread executes a chunk
  - when a thread finishes a chunk, it requests another one from the run-time system
- **The chunksize can be omitted**
  - when it is omitted, a chunksize of 1 is used

The Guided Schedule Type

- **Like for the dynamic schedule each thread executes a chunk**
  - when a thread finishes a chunk, it requests another one
  - as chunks are completed, the size of the new chunks decreases
  - goal: reduce work imbalance between threads
- **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

OpenMP “single” and “master” Directives

- **The single directive specifies that the enclosed region is executed only by a single thread of the team**
  - useful for calling library functions that are not thread-safe (e.g. I/O)

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

```
#pragma omp single
fprintf(output_file, “results”);
…
```

- **The master directive specifies a region that is to be executed only by the master thread**
  - it does not take any clauses
  - there is no implicit barrier after a master directive

OpenMP “barrier” Directive

- **The barrier directive synchronizes all treads in the team**
  - when reaching a barrier, a thread will wait at the point until all other threads have reached the same barrier
  - then, all threads resume executing the code following the barrier
  - useful for calling library functions that are not thread-safe (e.g. I/O)

```
#pragma omp barrier
```

Matrix-vector Multiplication

- **Reminder: Matrix-vector multiplication example from Pthreads chapter**
  - code is much simpler then Pthreads version
  - the problem with false-sharing for the 8 x 8,000,000 matrix still applies

```
#  pragma omp parallel for num_threads(thread_count)  \
      default(none) private(i, j) shared(A, x, y, m, n)
   for (i = 0; i < m; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
         y[i] += A[i][j]*x[j];
   }
```

	Matrix Dimension					
	8,000,000 × 8		**8000 × 8000**		**8 × 8,000,000**	
Threads	**Time**	**Eff.**	**Time**	**Eff.**	**Time**	**Eff.**
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275

Run-times and efficiencies of matrix-vector multiplication (times are in seconds)

Concluding Remarks

- **OpenMP is a standard for programming shared-memory systems**
  - controlled with directives, runtime-library functions and environment variables
  - OpenMP programs start multiple threads rather than multiple processes
  - Many OpenMP directives can be modified by clauses
- **A major problem in the development of shared memory programs is the possibility of race conditions**
  - OpenMP provides several mechanisms for insuring mutual exclusion in critical sections
- **OpenMP offers a variety of scheduling options.**
  - by default most systems use a block-partitioning of the iterations in a parallelized for loop
- **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**

Acknowledgements

- **Peter S. Pacheco / Elsevier**
  - for providing the lecture slides on which this presentation is based
- **OpenMP tutorial published by Lawrence Livermore National Lab**
  - https://computing.llnl.gov/tutorials/openMP/

Change log

- **1.1.1 (2017-11-27)**
  - update for winter term 2017/18
  - update outline slide 2
  - fix terminology and typos on slide 12, 14, 18, 19, 25, 28
- **1.1.0 (2017-07-13)**
  - fix typo on slide 44
- **1.0.1 (2017-01-30)**
  - fix typo on slide 30, 48
- **1.0.0 (2017-01-19)**
  - initial version of slides