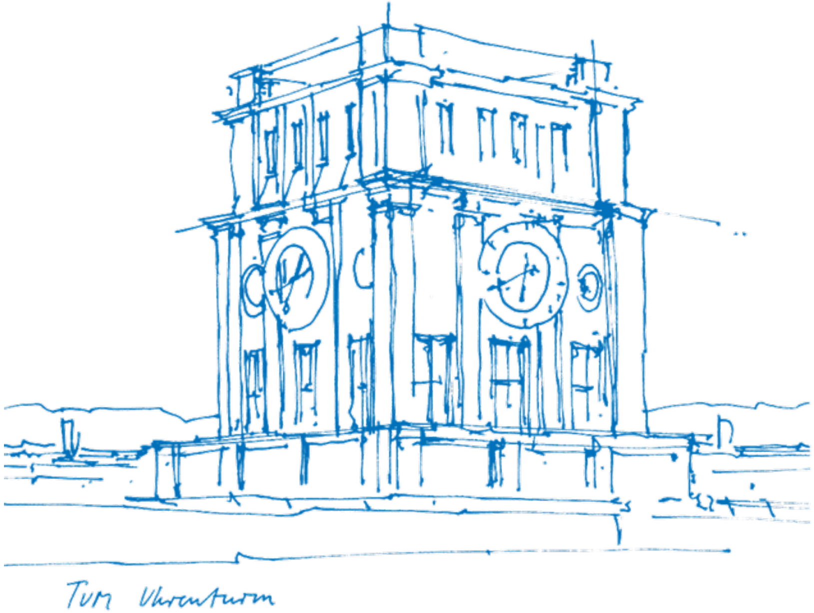
Parallel Programming Tutorial - OpenMP 2

M.Sc. Andreas Wilhelm
Technichal University Munich
June 12, 2017



Solution for Assignment 3

Solution for Assignment 3 (1/2)

- **Idea:** Statically distribute packs of words to `num_threads` tasks.
- Use a vector of futures to store the local histograms produced by the tasks.
- Use a local lambda function as task execution unit that gets a start and an end iterator.

```
void get_histogram(const std::vector<word_t>& words, histogram_t& histogram, int num_threads)
{
    int nWordsPerThread = words.size() / num_threads;
    std::vector<std::future<histogram_t>> futures;

    auto local_getHistogram = [](std::vector<word_t>::const_iterator firstIt,
                                 std::vector<word_t>::const_iterator lastIt) {
        histogram_t local_histogram {{0}};

        std::for_each(firstIt, lastIt, [&local_histogram](const word_t& word) {
                           int res = getNameIndex(word.data());
                           if (res != -1) local_histogram[res]++;
                       });

        return local_histogram;
    };
```

Solution for Assignment 3 (2/2)

- Compute start and end iterator position for each task.
- Use `std::async` to spawn the tasks and `get()` to retrieve the results.
- Do not forget to consider the remainder of the words.

```
for (int i = 0; i < num_threads; ++i) {
    const auto first = words.begin() + i       * nWordsPerThread;
    const auto last = words.begin() + (i + 1) * nWordsPerThread;

    futures.push_back(std::async(std::launch::async, local_getHistogram, first, last));
    if (i == (num_threads - 1))
        futures.push_back(std::async(local_getHistogram, last, words.end()));
}

for (auto& future : futures) {
    histogram_t local_histogram = future.get();
    for (int i = 0; i < NNAMES; ++i)
        histogram[i] += local_histogram[i];
}
}
```

Solution for Assignment 4

Solution for Assignment 4

- Use a parallel region with `num_threads` threads.
- Parallelize the outer loop for coarse grained parallelization.
- Use dynamic scheduling since the load is unbalanced.

```
void mandelbrot_draw(...)
{
  #pragma omp parallel num_threads(num_threads)
  {
    #pragma omp for schedule(dynamic)
    for (int i = 0; i < y_resolution; i++) {
      for (int j = 0; j < x_resolution; j++) {
      ...
        // no dependences between iterations
      ...
      }
    }
  }
}
```

OpenMP Sections

OpenMP Sections

```
#pragma omp sections <{clause, ...}>
{
   #pragma omp section
   <structured block>

   #pragma omp section
   <structured block>
}
```

- The sections directive contains a set of structured blocks that are executed by single threads of a team
- Each structured block is preceded by a section directive (except possibly the first one)
- The scheduling of the sections is implementation defined
- There is an implicit barrier at the end of a sections directive (unless `nowait`)
- Clauses: `private, firstprivate, lastprivate, reduction(identifier), nowait`

Nested Regions

```
// environmnet variable to set nested parallelism
OMP_NESTED
// library function to set/get nested parallelism
int omp_set_nested( int nested )
int omp_get_nested( void )
// limits/returns the number of maximal nested active parallel regions
int omp_set_max_active_levels( int max_levels )
int omp_get_max_active_levels( void )
// returns the number of current nesting level
int omp_get_level( void )
```

- Parallel regions and parallel sections may be arbitrarily nested inside each other
- If nested parallelism is disabled (default), the newly created team of threads will consist only of the encountering thread

Hint

- Take care of oversubscription when using nested parallelism.

OpenMP Tasks

Tasks (1/5)

Terminology

task	A specific instance of executable code and its data environment.
task region	A region consisting of all code encountered during the execution of a task.
explicit task	A task generated when a `task` construct is encountered.
implicit task	A task generated by an implicit parallel region.
tied task	A task that, when its task region is suspended, can be resumed only be the sam thread.
untied task	A task that, when its task region is suspended, can be resumed by any thread in the team.
undeferred task	A task for which execution is not deferred with respect to its generating task region.
included task	A task for which execution is sequentially included in the generating task region.
merged task	A task for which the data environment is the same as that of its generating task region.

Tasks (2/5)

```
#pragma omp task <{clause, ...}>
<structured block>
```

- Defines an explicit task, generated from the associated structured block.
- The encountering thread may immediately execute the task or defer it.
- Deferred tasks may be executed by any thread of the team.
- Tasks may be nested, but the task region of the inner task is not part of the task region of the outer task.
- A thread that encounters a task scheduling point (TSP) within a task may temporarily suspend this task.
- By default a task is tied to a thread (unless clause `untied`).

Tasks (3/5)

```
#pragma omp task <{clause, ...}>
<structured block>
```

Clauses (not exhaustive)

- `if (<scalar logical expression>)`
  if false, an undeferred task is generated
- `final (<scalar logical expression)`
  if true, the generated task and all child tasks are included (sequentialized) tasks are also final
- `default (private | firstprivate | shared | none)`
  default is firstprivate for tasks
- `mergeable`
  if the generated task is an undeferred or included task, the generation may generate a merged task
- `private, firstprivate, shared ( <list> )`
- `depend ( in | out | inout: list)`
  specifies dependencies across sibling tasks

Tasks (4/5)

```
#pragma omp taskyield
```

- Specifies that the current task can be suspended (implicit TSP)

```
#pragma omp taskwait
```

- Specifies a wait on the completion of child tasks of the current task (implict TSP)

```
#pragma omp taskgroup
```

- Specifies a wait on the completion of child tasks of the current task and their descendant tasks (implict TSP)

```
int omp_set_dynamic( int dynamic_threads )
```

- Enables or disables dynamic adjustment of number of threads available for tasks in subsequent parallel regions

Tasks (5/5)

Task Scheduling

Whenever a thread reaches a TSP, the implementation may perform a task switch, implied by the following locations:

- immediately following the generation of an explicit task
- after the completion of a task region
- in a taskyield region
- in a taskwait region
- at the end of a taskgroup region
- in an implicit or explicit barrier region
- ...

Example: Fibonacci Number (1/2)

- Application computes nth fibonacci number

```
int main(int argc, char** argv) {
  int n = 30;

  if(argc > 1)
    n= atoi(argv[1]);

  omp_set_num_threads(4);

  #pragma omp parallel shared(n)
  {
    #pragma omp single
    printf("fib(%d) = %d\n", n, fib(n));
  }
}
```

Example: Fibonacci Number (2/2)

```c
int fib(int n) {
  int i, j;

  if (n < 2) return n;

  #pragma omp task shared(i) firstprivate(n)
  i = fib(n - 1);

  #pragma omp task shared(j) firstprivate(n)
  j = fib(n - 2);

  #pragma omp taskwait
  return i + j;
}
```

Task Example: Fibonacci Number: Runtime

```
$ time ./fib 35
fib(35) = 9227465

real    0m9.785s
user    0m25.933s
sys     0m0.000s
```

Task Example: Fibonacci Number: final task

```c
#define T 30 // THRESHOLD

int fib(int n)
{
  int i, j;

  if (n < 2)
    return n;

  #pragma omp task shared(i) firstprivate(n) final(n > T)
  i = fib(n - 1);

  #pragma omp task shared(j) firstprivate(n) final(n > T)
  j = fib(n - 2);

  #pragma omp taskwait
  return i + j;
}
```

Task Example: Fibonacci Number: Runtime Final (GCC)

```
$ time ./fib_final 35
fib(35) = 9227465

real    0m0.392s
user    0m0.800s
sys     0m0.000s
```

Other directives (1/3)

```
#pragma omp single <{clause, ...}>
```

- The single directive specifies that the associated block is executed by only one thread (not necessarily the master)
- The other threads of the team wait at an implict barrier at the end of the single construct (unless `nowait`)
- Clauses: `private, firstprivate, copyprivate, nowait`

```
#pragma omp master <{clause, ...}>
```

- Same as single, but the thread is solely executed by the master thread
- Clauses: `private, firstprivate, copyprivate, nowait`

Other directives (2/3)

```
#pragma omp critical [<name>]
```

- Restricts the execution of the associated structured block to a single thread at a time
- An optional name may be used to identify the critical construct
- All critical constructs without a name use a default name

```
#pragma omp barrier
```

- Specifies an explicit barrier
- All threads of a team must execute the barrier region
- Includes an implicit task scheduling point

Other directives (3/3)

```
#pragma omp atomic [read | write | update | capture] [seq_cst]
<expression>
```

or

```
#pragma omp atomic [seq_cst]
<structured-block>
```

Example

```
#pragma omp atomic write
x = 41;

#pragma omp atomic
{
  v = x;
  x++;
}
```

- Ensures that a specific storage location is accessed atomically
- The expression reads|writes|read-writes|(read-writes + updates other variable) the storage location
- The structured block has two consecutive expressions
- Any atomic directive with a `seq_cst` clause forces a flush
- To avoid race conditions, all accesses to specific storage location must be protected with an atomic construct

Assignment 5

Assignment 5: `companytree`

Company Tree Algorithm

- The given algorithm computes the working hours for all employees in companies.
- It recursively traverses all hierarchy positions (Chairman -> {CIO, CTO, ..., Engineer}).
- At the end, a list of the emplyees with the most working hours are printed.

Part 1

- Parallelize the sequential company tree algorithm with OpenMP tasks
- Try to optimize it / reduce the overhead for tasking
- The goal is a speedup of > 10

Part 2

- Parallelize the sequential company tree algorithm with OpenMP sections
- Try to optimize it / reduce the overhead for nested parallelism
- The goal is a speedup of > 5

Assignment 5: companytree_seq.c

```
#include "companytree.h"

void traverse(tree *node, int numThreads)
{
   if(node != NULL){
      node->work_hours = compute_workHours(node->data);
      top_work_hours[node->id] = node->work_hours;

      traverse(node->right, numThreads);
      traverse(node->left, numThreads);
   }
}
```

Assignment 5: `companytree` with OpenMP - Provided Files

- Makefile
  - contains rules to build executables
  - available targets: parallel, sequential, unit_test, all (default), clean
  - 'mode=debug make [target]' to build debug version, use 'make clean' before
- main.c
  - main function - argument handling + call companytree algorithm
- companytree.h
  - Header file for companytree.c and companytree_*.c
- companytree.c
  - Defines the companytree logic
- ds.h / ds.c
  - Header and definition for the needed datastructures
- companytree_seq.c
  - Sequential version of `traverse()`.
- student/companytree_par.c
  - Implement the parallel version in this file

Assignment 5: `companytree` with OpenMP - Provided Files

- vis.h / vis.c
  - The visualization component
- unit_test.c
  - The unit tests that execute both the serial and parallel version to compare results.