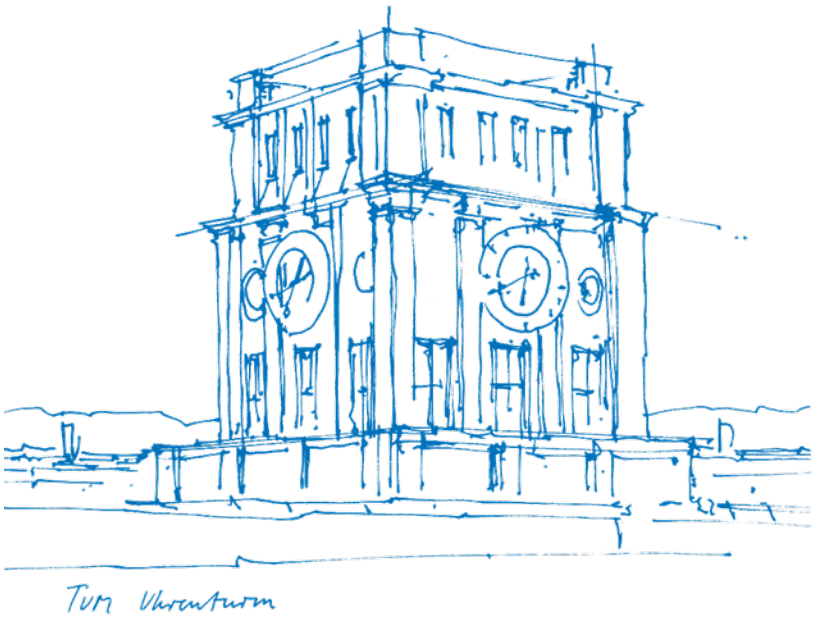
Parallel Programming Tutorial - OpenMP 2

M.Sc. Andreas Wilhelm

Technichal University Munich

May 30, 2016



Assignment 4: Possible Solution - `playGroups`

- Parallelize the outer loop to achieve coarse grained parallelism
- Set the number of threads to number of groups
  - use the clause `num_threads`
- Avoid race conditions by using copies of shared variables
  - use the clause `private` or
  - declare the variables inside the inner loop
- Use dynamic or guided scheduling to balance the threads
  - use the clause `schedule`

```c
void playGroups(team_t* teams) {
  static const int cNumTeamsPerGroup = NUMTEAMS / NUMGROUPS;
  int g, i, j, goalsI, goalsJ;

  #pragma omp parallel num_threads(NUMGROUPS)
  {
    #pragma omp for private (i, j, goalsI, goalsJ) schedule (dynamic)
    for (g = 0; g < NUMGROUPS; ++g) {
      for (i =  g * cNumTeamsPerGroup; i < (g+1) * cNumTeamsPerGroup; ++i) {
        for (j = (g+1) * cNumTeamsPerGroup - 1; j > i; --j) {

          // team i plays against team j in group g
        }
      }
    }
  }
}
```

Assignment 4: Possible Solution - `playFinalRound`

- Parallelize the loop
- Set the number of threads to number of games
  - use the clause `num_threads`
- Avoid race conditions by using copies of shared variables
  - use the clause `private` or
  - declare the variables inside the inner loop
- Use dynamic or guided scheduling to balance the threads
  - use the clause `schedule`

```c
void playFinalRound(int numGames, team_t** teams, team_t** successors){
  #pragma omp parallel num_threads(numGames)
  {
    team_t* team1;
    team_t* team2;
    int i, goals1 = 0, goals2 = 0;

    #pragma omp for schedule (dynamic)
    for (i = 0; i < numGames; ++i) {
     // play final match i
    }
  }
}
```

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, 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.

Tasks (1/4)

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

- Defines an explicit task, generated from the associated block
- The encountering thread may immediately execute 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 (2/4)

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

Clauses

- `if (<scalar logical expression>)`
  if false, an undeferred task is generated
- `final (<scalar logical expression)`
  if true, the generated task is a final (undeferred) task and all child tasks (included 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 (3/4)

```
#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 (4/4)

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 4: `familytree`

Family Tree Algorithm

- The given algorithm computes the IQ for all members in a family.
- It recursively traverses all 10 generations (child -> {mother, father}).
- At the end, all geniuses (IQ >= 140) are printed at the end.

Part 1

- Parallelize the sequential family tree algorithm with OpenMP tasks
- Try to optimize it / reduce the overhead for tasking
- The goal is a speedup of >= 3

Part 2

- Parallelize the sequential family tree algorithm with OpenMP sections
- Try to optimize it / reduce the overhead for nested parallelism
- The goal is a speedup of >= 2

Assignment 4: familytree_seq.c

```c
#include "familytree.h"

void traverse(tree* node, int numThreads){

   if(tree != NULL){
      node->IQ = compute_IQ(node->data);
      genius[node->id] = node->IQ;

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

      free(node); // node is allocated by fill()
   }
}
```

Assignment 4: `familytree` 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 familytree algorithm
- familytree.h
  - Header file for familytree.h and familytree_*.c
- familytree.c
  - Defines the familytree logic
- ds.h / ds.c
  - Header and definition for the needed datastructures
- familytree_seq.c
  - Sequential version of `traverse()`.
- student/familytree_par.c
  - Implement the parallel version in this file

Assignment 4: `familytree` 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.