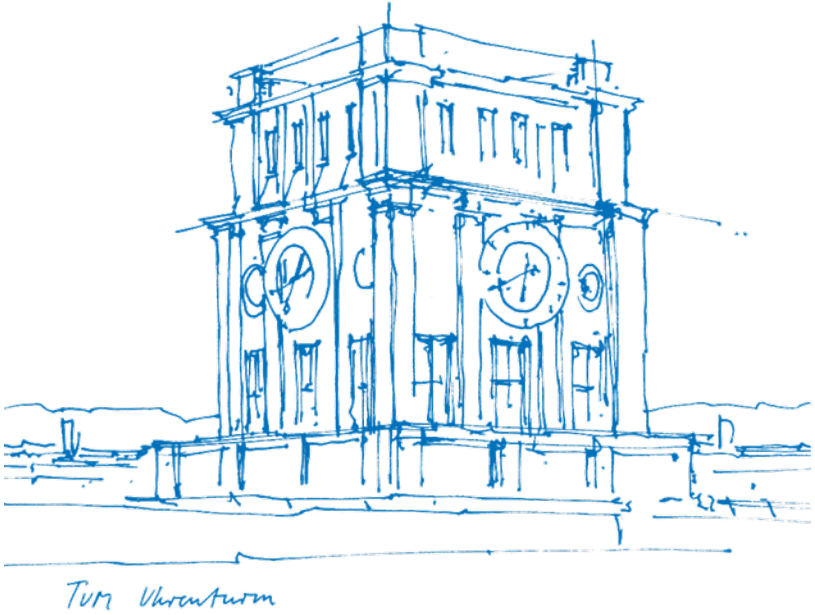
Parallel Programming Tutorial - More on OpenMP

Amir Raoofy, M.Sc.

Chair for Computer Architecture and Parallel Systems (Prof. Schulz)

Technichal University Munich

9. Mai 2018



Few organizational notes

Tutorial Schedule (short term)

- May 15th is the deadline for 3rd, 4th and 5th assignments.
- On May 16th we discuss the solutions for these assignments.
- On May 23rd we have the tutorial on dependency analysis and loop transformations
- On May 30th we start MPI

Recap from last tutorial on OpenMP

Quiz; how to create a team of four threads to print their ids

```
#include <iostream>
#include<omp.h>

int main(){

    int num_threads=4;
    omp_set_num_threads(num_threads);

    for (int i = 0; i < num_threads; i++)
    {
        std::cout << "My id is: "
                  << omp_get_thread_num() << std::endl;
    }
}
```

Quiz; how to create a team of four threads to print their ids

```cpp
#include <iostream>
#include<omp.h>

int main(){

    int num_threads=4;
    omp_set_num_threads(num_threads);

    for (int i = 0; i < num_threads; i++)
    {
        std::cout << "My id is: "
                  << omp_get_thread_num() << std::endl;
    }
}
```

```
./example1

My id is: 0
My id is: 0
My id is: 0
My id is: 0
```

Quiz (Cont.)

```
#include <iostream>
#include<omp.h>

int main(){

    int num_threads=4;
    omp_set_num_threads(num_threads);

    #pragma omp for
    for (int i = 0; i < num_threads; i++)
    {
        std::cout << "My id is: "
                  << omp_get_thread_num() << std::endl;
    }
}
```

Quiz (Cont.)

```cpp
#include <iostream>
#include<omp.h>

int main(){

    int num_threads=4;
    omp_set_num_threads(num_threads);

    #pragma omp for
    for (int i = 0; i < num_threads; i++)
    {
        std::cout << "My id is: "
                  << omp_get_thread_num() << std::endl;
    }
}
```

```
./example2

My id is: 0
My id is: 0
My id is: 0
My id is: 0
```

Quiz (Cont.)

```
#include <iostream>
#include<omp.h>

int main(){

    int num_threads=4;
    omp_set_num_threads(num_threads);

    #pragma omp parallel
    {
        for (int i = 0; i < num_threads; i++)
        {
            #pragma omp critical
            std::cout << "My id is: "
                      << omp_get_thread_num() << std::endl;
        }
    }
}
```

Quiz (Cont.)

```cpp
#include <iostream>
#include<omp.h>

int main(){

    int num_threads=4;
    omp_set_num_threads(num_threads);

    #pragma omp parallel
    {
        for (int i = 0; i < num_threads; i++)
        {
            #pragma omp critical
            std::cout << "My id is: "
                      << omp_get_thread_num() << std::endl;
        }
    }
}
```

```
./example3

My id is: 3
My id is: 0
My id is: 3
My id is: 0
My id is: 3
My id is: 0
My id is: 3
My id is: 0
My id is: 1
My id is: 1
My id is: 1
My id is: 1
My id is: 2
My id is: 2
My id is: 2
My id is: 2
```

Quiz (Cont.)

```cpp
#include <iostream>
#include<omp.h>

int main(){

    int num_threads=4;
    omp_set_num_threads(num_threads);

    #pragma omp parallel
    {
        #pragma omp parallel for
        for (int i = 0; i < num_threads; i++)
        {
            #pragma omp critical
            std::cout << "My id is: "
                      << omp_get_thread_num() << std::endl;
        }
    }
}
```

Quiz (Cont.)

```cpp
#include <iostream>
#include<omp.h>

int main(){

    int num_threads=4;
    omp_set_num_threads(num_threads);

    #pragma omp parallel
    {
        #pragma omp parallel for
        for (int i = 0; i < num_threads; i++)
        {
            #pragma omp critical
            std::cout << "My id is: "
                      << omp_get_thread_num() << std::endl;
        }
    }
}
```

```
./example4

My id is: 0
My id is: 0
My id is: 0
My id is: 0
My id is: 0
My id is: 0
My id is: 0
My id is: 0
My id is: 0
My id is: 0
My id is: 0
My id is: 0
My id is: 0
My id is: 0
My id is: 0
My id is: 0
```

Quiz (Cont.)

```
#include <iostream>
#include<omp.h>

int main(){

    int num_threads=4;
    omp_set_num_threads(num_threads);
    omp_set_nested(1);

    #pragma omp parallel
    {
        #pragma omp parallel for
        for (int i = 0; i < num_threads; i++)
        {
            #pragma omp critical
            std::cout << "My id is: "
                      << omp_get_thread_num() << std::endl;
        }
    }
}
```

Quiz (Cont.)

```cpp
#include <iostream>
#include<omp.h>

int main(){

    int num_threads=4;
    omp_set_num_threads(num_threads);
    omp_set_nested(1);

    #pragma omp parallel
    {
        #pragma omp parallel for
        for (int i = 0; i < num_threads; i++)
        {
            #pragma omp critical
            std::cout << "My id is: "
                      << omp_get_thread_num() << std::endl;
        }
    }
}
```

```
./example5

My id is: 1
My id is: 0
My id is: 2
My id is: 3
My id is: 1
My id is: 2
My id is: 0
My id is: 1
My id is: 1
My id is: 0
My id is: 3
My id is: 2
My id is: 3
My id is: 0
My id is: 3
My id is: 2
```

Quiz (Cont.)

```cpp
#include <iostream>
#include<omp.h>

int main(){

    int num_threads=4;
    omp_set_num_threads(num_threads);

    #pragma omp parallel
    {
        #pragma omp for
        for (int i = 0; i < num_threads; i++)
        {
            #pragma omp critical
            std::cout << "My id is: "
                      << omp_get_thread_num() << std::endl;
        }
    }
}
```

```
./example6

My id is: 0
My id is: 1
My id is: 2
My id is: 3
```

Quiz (Cont.)

```
#include <iostream>
#include<omp.h>

int main(){

    int num_threads=4;
    omp_set_num_threads(num_threads);

    #pragma omp parallel for
    for (int i = 0; i < num_threads; i++)
    {
        #pragma omp critical
        std::cout << "My id is: "
                  << omp_get_thread_num() << std::endl;
    }
}
```

```
./example7

My id is: 2
My id is: 0
My id is: 1
My id is: 3
```

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.

Example: Traverse a binary tree

```
struct node
{
    struct node *left, *right;
    int key;
    node(int k):key(k){}
};

void traverse(struct node *p)
{
    if (p->left != NULL)
        traverse(p->left);

    if (p->right != NULL)
        traverse(p->right);

    process(p);
}
```

```
void process(struct node *p){
    usleep(1000000);
    std::cout << "element with key: "
              << p->key << " is processed"
              << std::endl;
}

int main(int argc, char *argv[])
{
    struct node *tree = new struct node(0);
    tree->left = new struct node(1);
    tree->right = new struct node(2);
    tree->left->left  = new struct node(3);
    tree->left->right = new struct node(4);
    tree->right->left  = new struct node(5);
    tree->right->right = new struct node(6);

    traverse(tree);
    return 0;
}
```

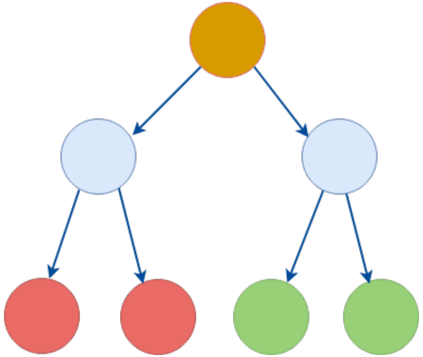
Example: Traverse a binary tree (Cont.)

```
void traverse(struct node *p)
{
    #pragma omp parallel
    {
        #pragma omp sections
        {

            #pragma omp section
            {
                if (p->left != NULL)
                    traverse(p->left);
            }

            #pragma omp section
            {
                if (p->right != NULL)
                    traverse(p->right);
            }
        }
    }
    process(p);
}
```

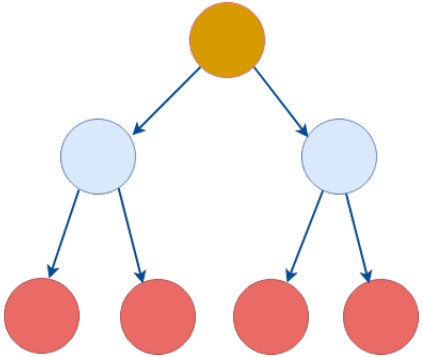
```
void process(struct node *p){
    usleep(1000000);
    #pragma omp critical
    std::cout << "element with key: "
              << p->key << " is processed"
              << std::endl;
}
int main(int argc, char *argv[])
{
    struct node *tree = new struct node(0);
    tree->left = new struct node(1);
    tree->right = new struct node(2);
    tree->left->left  = new struct node(3);
    tree->left->right = new struct node(4);
    tree->right->left  = new struct node(5);
    tree->right->right = new struct node(6);

    omp_set_nested(1);
    omp_set_max_active_levels(2);

    traverse(tree);
    return 0;
}
```

Where is the parallelism in tree traversal?

- omp_set_max_active_levels(1);



- omp_set_max_active_levels(2);

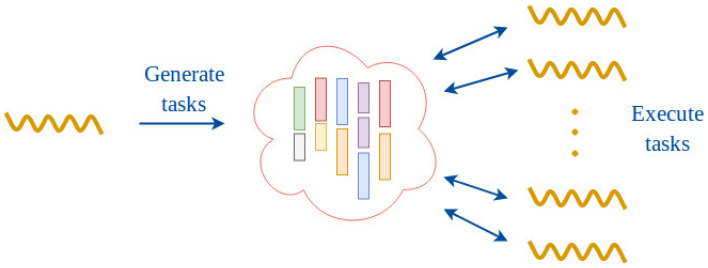


OpenMP Tasks

OpenMP Tasks

Why Tasks?

- We don't always deal with simple for loops for parallelization
- We don't always deal with simple data structures like arrays
- Some times we don't know the length of the loops at compile time e.g., while loop
- Some times we deal with unknown number of parallel sections
- We need to deal with parallelization of recursive algorithms
- It is possible without tasks (OpenMP 3.0) but it is not pretty



Task semantics

Terminology

task A specific instance of executable code and its data environment and ICVs.

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 by the same 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.

Task semantics (Cont.)

The **task pragma** can be used to explicitly define a task.

Use the task pragma when you want to identify a block of code to be executed in parallel with the code outside the task region. The task pragma can be useful for parallelizing irregular algorithms such as pointer chasing or recursive algorithms. The task directive takes effect only if you specify the SMP compiler option.

```
#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`).

Task syntax

```
#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

Task Scheduling Points (TSPs)

```
#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

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 1: Hello world using tasks

```cpp
#include <iostream>
#include <omp.h>

int main(int argc, char *argv[])
{
        #pragma omp parallel
        {
            #pragma omp task
            std::cout << "Hello World from task"
                      << std::endl;
        }
        return 0;
}
```

```
OMP_NUM_THREADS=4 ./example1
```

```
Hello World from task
Hello World from task
Hello World from task
Hello World from task
```

Example 2: Which threads execute the tasks

```cpp
#include <iostream>
#include <omp.h>

int main(int argc, char *argv[])
{
        #pragma omp parallel
        {
            #pragma omp task
            {
                #pragma omp critical
                std::cout << "Hello World from task,\
                              executed by thread: "
                          << omp_get_thread_num()
                          << std::endl;
            }
        }
        return 0;
}
```

directive identifies a section of code that must be executed by a single thread at a time.

OMP_NUM_THREADS=4 ./example2

```
Hello World from task, executed by thread: 0
Hello World from task, executed by thread: 3
Hello World from task, executed by thread: 2
Hello World from task, executed by thread: 1
```

or

```
Hello World from task, executed by thread: 0
Hello World from task, executed by thread: 1
Hello World from task, executed by thread: 2
Hello World from task, executed by thread: 0
```

or ...

Example 3: Using singe thread to create tasks

```
int main(int argc, char *argv[])
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            for (int t = 0; t < omp_get_num_threads(); t++)
            {
                #pragma omp task
                {
                    #pragma omp critical
                    std::cout << "Hello World from task,\
                                    executed by thread: "
                              << omp_get_thread_num()
                              << std::endl;
                }
            }
        }
    }
    return 0;
}
```

The **omp single** directive identifies a section of code that must be run by a single available thread

```
OMP_NUM_THREADS=4 ./example3

Hello World from task, executed by thread: 2
Hello World from task, executed by thread: 1
Hello World from task, executed by thread: 2
Hello World from task, executed by thread: 0
```

- Only one thread creates the tasks
- Unlike the previous example where all threads created tasks
- Created tasks can be nested and are scheduled to be executed by the available threads

Example 4: List traversal

```cpp
void process_element(int &elem){
        usleep(1000000);
        std::cout << elem << std::endl;
}

void traverse_list(std::forward_list<int> &l){
        for (auto it = l.begin(); it != l.end() ; it++) {
                process_element(*it);
        }
}

int main(int argc, char *argv[])
{
        std::forward_list<int> l;
        l.assign({0,1,2,3,4,5,6,7,8,9});

        traverse_list(l);

        return 0;
}
```

```
time ./example4

0
1
2
3
4
5
6
7
8
9

real 0m10.006s
```

Example 4: List traversal (Cont.)

```
void process_element(int &elem){
        usleep(1000000);
        #pragma omp critical
        std::cout << elem << std::endl;
}

void traverse_list(std::forward_list<int> &l){
        #pragma omp parallel
        {
                #pragma omp single
                for (auto it = l.begin(); it != l.end() ; it++) {
                        #pragma omp task
                        process_element(*it);
                }
        }
}
```

```
time OMP_NUM_THREADS=4
./example4

0
1
2
3
4
5
6
7
8
9

real 0m3.015s
```

Example 5: Fibonacci Number

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

    if (n < 2) return n;

    i = fib(n - 1);
    j = fib(n - 2);

    return i + j;
}
```

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

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

    printf("fib(%d) = %d\n", n, fib(n));

}
```

Example 5: Fibonacci Number (Cont.)

```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;
}
```

```c
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 5: Fibonacci Number, Runtime

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

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

Example 5: 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;
}
```

Declares the scope of the data variables in list to be private to each thread. Each new private object is initialized with the value of the original variable as if there was an implied declaration within the statement block. Data variables in list are separated by commas.

Use the **taskwait pragma** to specify a wait for child tasks to be completed that are generated by the current task

Example 5: Fibonacci Number, Runtime Final (GCC)

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

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

Other directives

```
#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 (Cont.)

```
#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 (Cont.)

```
#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 a shared storage location must be protected with an atomic construct

Assignment 5

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.

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

- You will have two weeks for this assignment.

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 (Cont.)

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