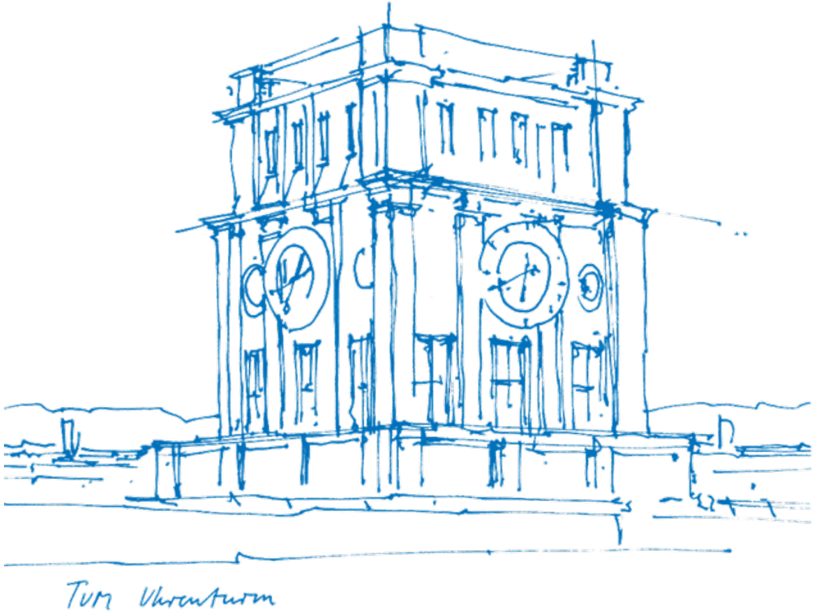
Parallel Programming Tutorial - OpenMP Wrap-Up

Amir Raoofy, M.Sc.

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

Technical University of Munich

16. Mai 2018



Few organizational notes

Organization

- Deadline for assignment 5 is extended; May 22nd
- We have an optional assignment on SIMD (will be released next week)
  - Based on guest lecture by Dr. Michael Klemm
- On June 25th we will have a lecture on Optimization of sequential programs by M.Sc. Alexis Engelke
  Topics: (tentative)
  - Compiler optimizations
  - Floating-point optimizations
  - (Auto-)Vectorization
  - Cache optimizations
  - Eventually profiling

OpenMP Wrap-Up

Nested parallel regions revisited

```
#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;
        }
    }
}
```

Nested parallel regions revisited

```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
```

Nested parallel regions revisited (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;
        }
    }
}
```

Nested parallel regions revisited (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; What is the problem with this program?

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

int main(){

    int id;
    #pragma omp parallel num_threads(4)
    {
        id = omp_get_thread_num();
        #pragma omp critical
        std::cout << "My id is: " << id << std::endl;
    }

}
```

Quiz; What is the problem with this program?

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

int main(){

    int id;
    #pragma omp parallel num_threads(4)
    {
        id = omp_get_thread_num();
        #pragma omp critical
        std::cout << "My id is: " << id << std::endl;
    }

}
```

./example

Quiz; What is the problem with this program?

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

int main(){

    int id;
    #pragma omp parallel num_threads(4)
    {
        id = omp_get_thread_num();
        #pragma omp critical
        std::cout << "My id is: " << id << std::endl;
    }

}
```

```
./example

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

Quiz; What is the problem with this program?

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

int main(){

    int id;
    #pragma omp parallel num_threads(4)
    {
        id = omp_get_thread_num();
        #pragma omp critical
        std::cout << "My id is: " << id << std::endl;
    }

}
```

```
./example

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

```
./example
```

Quiz; What is the problem with this program?

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

int main(){

    int id;
    #pragma omp parallel num_threads(4)
    {
        id = omp_get_thread_num();
        #pragma omp critical
        std::cout << "My id is: " << id << std::endl;
    }

}
```

```
./example

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

```
./example

My id is: 2
My id is: 2
My id is: 0
My id is: 0
```

Quiz; What is the problem with this program? (Cont.)

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

int main(){

    int id;
    #pragma omp parallel num_threads(4) private(id)
    {
        id = omp_get_thread_num();
        #pragma omp critical
        std::cout << "My id is: " << id << std::endl;
    }

}
```

Quiz; What is the problem with this program? (Cont.)

./example

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

int main(){

    int id;
    #pragma omp parallel num_threads(4) private(id)
    {
        id = omp_get_thread_num();
        #pragma omp critical
        std::cout << "My id is: " << id << std::endl;
    }

}
```

Quiz; What is the problem with this program? (Cont.)

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

int main(){

    int id;
    #pragma omp parallel num_threads(4) private(id)
    {
        id = omp_get_thread_num();
        #pragma omp critical
        std::cout << "My id is: " << id << std::endl;
    }

}
```

private()
Declares the scope of the data variables in list to be private to each thread. Data variables in list are separated by commas.

```
./example

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

Quiz; Data scoping

```c
int main (){
    int a =1, b =2, c =3;
    #pragma omp parallel num_threads(4) private(b) firstprivate(c)
    {
        #pragma omp critical
        {
            a++;

        }
        #pragma omp barrier
        if (omp_get_thread_num()==0){
            printf("a: %d\n", a);
            printf("b: %d\n", b);
            printf("c: %d\n", c);
        }
    }

    printf("a: %d\n", a);
    printf("b: %d\n", b);
    printf("c: %d\n", c);
    return 0;
}
```

The **omp barrier** directive identifies a synchronization point at which threads in a parallel region will wait until all other threads in that section reach the same point. Statement execution past the

Quiz; Data scoping

```c
int main (){
    int a =1, b =2, c =3;
    #pragma omp parallel num_threads(4) private(b) firstprivate(c)
    {
        #pragma omp critical
        {
            a++;  // -> shared
            b++;

        }
        #pragma omp barrier
        if (omp_get_thread_num()==0){
            printf("a: %d\n", a);
            printf("b: %d\n", b);
            printf("c: %d\n", c);
        }
    }

    printf("a: %d\n", a);
    printf("b: %d\n", b);
    printf("c: %d\n", c);
    return 0;
}
```

Quiz; Data scoping

```c
int main (){
    int a =1, b =2, c =3;
    #pragma omp parallel num_threads(4) private(b) firstprivate(c)
    {
        #pragma omp critical
        {
            a++;  // -> shared
            b++;  // -> private
            c++;
        }
        #pragma omp barrier
        if (omp_get_thread_num()==0){
            printf("a: %d\n", a);
            printf("b: %d\n", b);
            printf("c: %d\n", c);
        }
    }

    printf("a: %d\n", a);
    printf("b: %d\n", b);
    printf("c: %d\n", c);
    return 0;
}
```

Quiz; Data scoping

```
int main (){
    int a =1, b =2, c =3;
    #pragma omp parallel num_threads(4) private(b) firstprivate(c)
    {
        #pragma omp critical
        {
            a++;  // -> shared
            b++;  // -> private
            c++;  // -> firstprivate
        }
        #pragma omp barrier
        if (omp_get_thread_num()==0){
            printf("a: %d\n", a);
            printf("b: %d\n", b);
            printf("c: %d\n", c);
        }
    }

    printf("a: %d\n", a);
    printf("b: %d\n", b);
    printf("c: %d\n", c);
    return 0;
}
```

Quiz; Data scoping

```
int main (){
    int a =1, b =2, c =3;
    #pragma omp parallel num_threads(4) private(b) firstprivate(c)
    {
        #pragma omp critical
        {
            a++;  // -> shared
            b++;  // -> private
            c++;  // -> firstprivate
        }
        #pragma omp barrier
        if (omp_get_thread_num()==0){
            printf("a: %d\n", a);
            printf("b: %d\n", b);
            printf("c: %d\n", c);
        }
    }

    printf("a: %d\n", a);
    printf("b: %d\n", b);
    printf("c: %d\n", c);
    return 0;
}
```

./example

Quiz; Data scoping

```
int main (){
    int a =1, b =2, c =3;
    #pragma omp parallel num_threads(4) private(b) firstprivate(c)
    {
        #pragma omp critical
        {
            a++;  // -> shared
            b++;  // -> private
            c++;  // -> firstprivate
        }
        #pragma omp barrier
        if (omp_get_thread_num()==0){
            printf("a: %d\n", a);
            printf("b: %d\n", b);
            printf("c: %d\n", c);
        }
    }

    printf("a: %d\n", a);
    printf("b: %d\n", b);
    printf("c: %d\n", c);
    return 0;
}
```

```
./example


a: 5
```

Quiz; Data scoping

```
int main (){
    int a =1, b =2, c =3;
    #pragma omp parallel num_threads(4) private(b) firstprivate(c)
    {
        #pragma omp critical
        {
            a++;  // -> shared
            b++;  // -> private
            c++;  // -> firstprivate
        }
        #pragma omp barrier
        if (omp_get_thread_num()==0){
            printf("a: %d\n", a);
            printf("b: %d\n", b);
            printf("c: %d\n", c);
        }
    }

    printf("a: %d\n", a);
    printf("b: %d\n", b);
    printf("c: %d\n", c);
    return 0;
}
```

```
./example


a: 5
b: ?
```

Quiz; Data scoping

```c
int main (){
    int a =1, b =2, c =3;
    #pragma omp parallel num_threads(4) private(b) firstprivate(c)
    {
        #pragma omp critical
        {
            a++;  // -> shared
            b++;  // -> private
            c++;  // -> firstprivate
        }
        #pragma omp barrier
        if (omp_get_thread_num()==0){
            printf("a: %d\n", a);
            printf("b: %d\n", b);
            printf("c: %d\n", c);
        }
    }

    printf("a: %d\n", a);
    printf("b: %d\n", b);
    printf("c: %d\n", c);
    return 0;
}
```

```
./example


a: 5
b: ?
c: 4
```

Quiz; Data scoping

```
int main (){
    int a =1, b =2, c =3;
    #pragma omp parallel num_threads(4) private(b) firstprivate(c)
    {
        #pragma omp critical
        {
            a++;  // -> shared
            b++;  // -> private
            c++;  // -> firstprivate
        }
        #pragma omp barrier
        if (omp_get_thread_num()==0){
            printf("a: %d\n", a);
            printf("b: %d\n", b);
            printf("c: %d\n", c);
        }
    }

    printf("a: %d\n", a);
    printf("b: %d\n", b);
    printf("c: %d\n", c);
    return 0;
}
```

```
./example


a: 5
b: ?
c: 4
a: 5
```

Quiz; Data scoping

```c
int main (){
    int a =1, b =2, c =3;
    #pragma omp parallel num_threads(4) private(b) firstprivate(c)
    {
        #pragma omp critical
        {
            a++;  // -> shared
            b++;  // -> private
            c++;  // -> firstprivate
        }
        #pragma omp barrier
        if (omp_get_thread_num()==0){
            printf("a: %d\n", a);
            printf("b: %d\n", b);
            printf("c: %d\n", c);
        }
    }

    printf("a: %d\n", a);
    printf("b: %d\n", b);
    printf("c: %d\n", c);
    return 0;
}
```

```
./example


a: 5
b: ?
c: 4
a: 5
b: 2
```

Quiz; Data scoping

```c
int main (){
    int a =1, b =2, c =3;
    #pragma omp parallel num_threads(4) private(b) firstprivate(c)
    {
        #pragma omp critical
        {
            a++;  // -> shared
            b++;  // -> private
            c++;  // -> firstprivate
        }
        #pragma omp barrier
        if (omp_get_thread_num()==0){
            printf("a: %d\n", a);
            printf("b: %d\n", b);
            printf("c: %d\n", c);
        }
    }

    printf("a: %d\n", a);
    printf("b: %d\n", b);
    printf("c: %d\n", c);
    return 0;
}
```

```
./example


a: 5
b: ?
c: 4
a: 5
b: 2
c: 3
```

Quiz; Task data scoping

```
int a=1;
void parallel_function()
{
    int b=2, c=3;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d=4;
        #pragma omp task
        {
            int e=5;
            a





        }
    }
}
```

Quiz; Task data scoping

```
int a=1;
void parallel_function()
{
    int b=2, c=3;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d=4;
        #pragma omp task
        {
            int e=5;
            a  // shared




        }
    }
}
```

Quiz; Task data scoping

```
int a=1;
void parallel_function()
{
    int b=2, c=3;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d=4;
        #pragma omp task
        {
            int e=5;
            a  // shared          -> a=1
            b




        }
    }
}
```

Quiz; Task data scoping

```
int a=1;
void parallel_function()
{
    int b=2, c=3;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d=4;
        #pragma omp task
        {
            int e=5;
            a  // shared         -> a=1
            b  // firstprivate



        }
    }
}
```

Quiz; Task data scoping

```
int a=1;
void parallel_function()
{
    int b=2, c=3;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d=4;
        #pragma omp task
        {
            int e=5;
            a  // shared         -> a=1
            b  // firstprivate   -> b=?
            c



        }
    }
}
```

Quiz; Task data scoping

```
int a=1;
void parallel_function()
{
    int b=2, c=3;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d=4;
        #pragma omp task
        {
            int e=5;
            a  // shared        -> a=1
            b  // firstprivate  -> b=?
            c  // shared



        }
    }
}
```

Quiz; Task data scoping

```
int a=1;
void parallel_function()
{
    int b=2, c=3;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d=4;
        #pragma omp task
        {
            int e=5;
            a  // shared         -> a=1
            b  // firstprivate   -> b=?
            c  // shared         -> c=3
            d

        }
    }
}
```

Quiz; Task data scoping

```
int a=1;
void parallel_function()
{
    int b=2, c=3;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d=4;
        #pragma omp task
        {
            int e=5;
            a  // shared         -> a=1
            b  // firstprivate   -> b=?
            c  // shared         -> c=3
            d  // firstprivate

        }
    }
}
```

Quiz; Task data scoping

```
int a=1;
void parallel_function()
{
    int b=2, c=3;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d=4;
        #pragma omp task
        {
            int e=5;
            a  // shared         -> a=1
            b  // firstprivate   -> b=?
            c  // shared         -> c=3
            d  // firstprivate   -> d=4
            e
        }
    }
}
```

Quiz; Task data scoping

```
int a=1;
void parallel_function()
{
    int b=2, c=3;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d=4;
        #pragma omp task
        {
            int e=5;
            a  // shared          -> a=1
            b  // firstprivate    -> b=?
            c  // shared          -> c=3
            d  // firstprivate    -> d=4
            e  // private
        }
    }
}
```

Quiz; Task data scoping

```
int a=1;
void parallel_function()
{
    int b=2, c=3;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d=4;
        #pragma omp task
        {
            int e=5;
            a  // shared         -> a=1
            b  // firstprivate   -> b=?
            c  // shared         -> c=3
            d  // firstprivate   -> d=4
            e  // private        -> e=5
        }
    }
}
```

Quiz; Coarse-grained parallelization

```c
#define N 10000
#define ITER 100
double A[N + 2][N + 2];

int main(int argc, char **argv)
{

    for (int i = 0; i < N + 2; i++)          // Initialization
        for (int j = 0; j < N + 2; j++)
            A[i][j] = 0.0;

    for (int i = 0; i < N + 2; i++){         // Boundary conditions
        A[i][0] = 1.0; A[i][N + 2] = 1.0;
    }

    for (int n = 0; n < 100; n++){           // Main iteration loop

        for (int i = 1; i < N + 1; i++)
            for (int j = 1; j < N + 1; j++)
                A[i][j] = (A[i+1][j+1] + A[i-1][j-1] + A[i+1][j-1] + A[i-1][j+1])/4;
    }
    return 0;
}
```

Quiz; Coarse-grained parallelization

```c
#define N 10000
#define ITER 100
double A[N + 2][N + 2];

int main(int argc, char **argv)
{

    for (int i = 0; i < N + 2; i++)              // Initialization
        for (int j = 0; j < N + 2; j++)
            A[i][j] = 0.0;

    for (int i = 0; i < N + 2; i++){             // Boundary conditions
        A[i][0] = 1.0; A[i][N + 2] = 1.0;
    }

    for (int n = 0; n < 100; n++){               // Main iteration loop
        #pragma omp parallel for                 // Coarse-grained parallelization
        for (int i = 1; i < N + 1; i++)
            for (int j = 1; j < N + 1; j++)
                A[i][j] = (A[i+1][j+1] + A[i-1][j-1] + A[i+1][j-1] + A[i-1][j+1])/4;
    }
    return 0;
}
```

Quiz; Coarse-grained parallelization

```c
#define N 10000
#define ITER 100
double A[N + 2][N + 2];

int main(int argc, char **argv)
{
    #pragma omp parallel for                 // First touch
    for (int i = 0; i < N + 2; i++)          // Initialization
        for (int j = 0; j < N + 2; j++)
            A[i][j] = 0.0;

    for (int i = 0; i < N + 2; i++){         // Boundary conditions
        A[i][0] = 1.0; A[i][N + 2] = 1.0;
    }

    for (int n = 0; n < 100; n++){           // Main iteration loop
        #pragma omp parallel for             // Coarse-grained parallelization
        for (int i = 1; i < N + 1; i++)
            for (int j = 1; j < N + 1; j++)
                A[i][j] = (A[i+1][j+1] + A[i-1][j-1] + A[i+1][j-1] + A[i-1][j+1])/4;
    }
    return 0;
}
```

Typical patterns that come up in parallel programming

- Loop parallelization (Worksharing)
  - Parallelize the for loops that are time consuming in the code
  - Make sure the loops are parallelizable (dependency analysis)
  - Put the pragmas and take care of the data attributes

- Example:

```
// Initialization ...

for (int n = 0; n < 100; n++){
    #pragma omp parallel for
    for (int i = 1; i < N + 1; i++)
        for (int j = 1; j < N + 1; j++)
            A[i][j] = (A[i+1][j+1] + A[i-1][j-1] + A[i+1][j-1] + A[i-1][j+1])/4;
}
```

Typical patterns that come up in parallel programming (Cont.)

- Divide and conquer and unstructured parallelism (Tasking)
  - Split the problem into subproblems
  - Solver the subproblems in parallel
  - Fits the Tasking in OpenMP (v3 and later)

- Example:

```c
struct node
{
    struct node* left;
    struct node* right;
};

void traverse( struct node*p ) {
    if(p->left)
        #pragma omp task
        traverse(p->left);
    if(p->right)
        #pragma omp task
        traverse(p->right);
    process(p);
}
```

```c
// main

#pragma omp parallel
{
    #pragma omp single
    traverse(root);
}
```

Solution for Assignment 3

Solution for Assignment 3

```cpp
template <typename SrcView, typename DstView>
void x_gradient(const SrcView &src, const DstView &dst, int num_threads)
{
    int start = 0;
    int chunk_size = 16;
    std::mutex mtx;

    std::vector<std::thread> threads;

    for (int i = 0; i < num_threads; ++i)
    {
        threads.push_back(std::thread(x_gradient_kernel<SrcView, DstView>, \
                                      std::ref(src), std::ref(dst), \
                                      std::ref(start), std::ref(chunk_size), std::ref(mtx)));
    }

    for (auto &th : threads)
        th.join();
}
```

Solution for Assignment 3 (Cont.)

```
template <typename SrcView, typename DstView>
void *x_gradient_kernel(const SrcView &src, const DstView &dst,
                        int &start, int &chunk_size, std::mutex &mtx)
{
    typedef typename channel_type<DstView>::type dst_channel_t;
    int local_start;

    while (true)
    {
        // next slide
    }
}
```

Solution for Assignment 3 (Cont.)

```
    while (true)
    {
        mtx.lock();
        local_start = start;
        if (src.height() - start < 1) {
            mtx.unlock(); break;
        }

        if (src.height() - local_start < chunk_size)
            chunk_size = src.height() - local_start;

        start += chunk_size;
        mtx.unlock();

        for (int y = local_start; y < local_start + chunk_size; ++y) {
            typename SrcView::x_iterator src_it = src.row_begin(y);
            typename DstView::x_iterator dst_it = dst.row_begin(y);

            for (int x = 1; x < src.width() - 1; ++x)
                static_transform(src_it[x - 1], src_it[x + 1], dst_it[x],
                                 halfdiff_cast_channels<dst_channel_t>());
        }
    }
```

Solution for Assignment 4

Solution for Assignment 4

```
template <typename SrcView, typename DstView>
void x_gradient(const SrcView& src, const DstView& dst, int num_threads) {
    typedef typename channel_type<DstView>::type dst_channel_t;

    omp_set_num_threads(num_threads);

    #pragma omp parallel for schedule (dynamic,5)
    for (int y=0; y<src.height(); ++y) {
        typename SrcView::x_iterator src_it = src.row_begin(y);
        typename DstView::x_iterator dst_it = dst.row_begin(y);

        for (int x=1; x<src.width()-1; ++x) {
            static_transform(src_it[x-1], src_it[x+1], dst_it[x],
                             halfdiff_cast_channels<dst_channel_t>());
        }
    }
}
```

Hints for Assignment 5

Hints for Assignment 5

- If you use sections:
  - Pay attention to the recursive structure of `traverse`
  - Make sure that nesting is enabled
  - Stop nesting at a specific level to prevent creating so many parallel regions

- If you use tasks:
  - Again, pay attention to the recursive structure of `traverse`
  - Make sure that you create appropriate number of tasks
  - Try to restrict the number of created tasks using the appropriate clauses

Assignment 6 - Laplace 2D

Assignment 6 - Laplace 2D

- 2d Laplace equation with fixed boundaries
- Problem domain is unit square with uniform mesh
- Finite differences are used for the discretization
- We use Jacobi iterative method to solve the equation
- Look into the code and find the bottlenecks
- Use OpenMP to parallelize the solver
- You need to get a speedup of 16 on our server with 32 logical cores
- The server has 2 NUMA nodes each with 8 cores
- Pay attention to data locality on the cores

Assignment 6 - Laplace 2D - 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 initialization of arrays and main iteration loop
- laplace.c
  - implementations
- laplace.h
  - Header and definitions for the arrays
- laplace_seq.c
  - Sequential version of `time_step()`.
- student/laplace_par.c
  - Implement the parallel version in this file

Assignment 6 - Laplace 2D - 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.