Outline:

- OpenMP — the Basics
- Coffee Break
- OpenMP — the Advanced Story
Exploiting Loop-Level Parallelism with OpenMP

So far: Limited to individual loops:

- No relationship between subsequent parallelizable loops.
- Parallel execution environment repeatedly set up and terminated.
- Frequent barrier synchronizations
- Limited scope for optimization

Wanted: Larger parallel sections

- Containing several parallel loops
- Containing other work sharing constructs
- Barrier synchronization only as necessary
- Overlapping of different parallel activities
Example: Mandelbrot with Dithering

Sequential code:

```c
for (i=0; i<M; i++) {
    for (j=0; j<N; j++) {
        x = (double) i / (double) M;
        y = (double) j / (double) N;
        depth[i,j] = mandelval( x, y, max);
    }
}

for (i=0; i<M; i++) {
    for (j=1; j<N-1; j++) {
        dith[i,j] = 0.5 * depth[i,j]
                    + 0.25 * depth[i,j-1]
                    + 0.25 * depth[i,j+1];
    }
}
```

Outline:

1. Compute Mandelbrot picture
2. Perform dithering step
Mandelbrot with Dithering

Loop-parallelized code:

```c
#pragma omp parallel for private(i,j,x,y)
for (i=0; i<M; i++) {
    for (j=0; j<N; j++) {
        x = (double) i / (double) M;
        y = (double) j / (double) N;
        depth[i,j] = mandelval( x, y, max);
    }
}
```

```c
#pragma omp parallel for private(i,j)
for (i=0; i<M; i++) {
    for (j=1; j<N-1; j++) {
        dith[i,j] = 0.5 * depth[i,j]
                    + 0.25 * depth[i,j-1]
                    + 0.25 * depth[i,j+1];
    }
}
```

Disadvantage:

- Between the two parallel loops:
  - All threads hit synchronization barrier.
  - Worker threads are terminated.
  - Worker threads are re-created.

- **Avoidable overhead !!**
- Way out: Decoupling of
  - parallelization
  - work sharing
Introducing the `parallel`-directive:

```c
#pragma omp parallel
{
    printf( "Hello world says thread %d.\n", omp_get_thread_num());
}
```

**Effect:**

![Diagram showing parallel region execution]

Compiler Directives: OpenMP

Clemens Grelck
Computer Systems Architecture
University of Amsterdam
SPMD-Style Parallel Regions

Introducing the parallel-directive:

```c
#pragma omp parallel
{
    ...
}
```

Meaning:

- The entire code block following the `parallel`-directive is executed by all threads concurrently.
- This includes:
  - creation of worker threads,
  - SPMD-style code execution,
  - barrier synchronization,
  - termination of worker threads.
**SPMD-Style Parallel Regions**

Introducing the `parallel-directive`:

```plaintext
#pragma omp parallel
{
    ...
}
```

Available clauses:

- `private (list of variables)`
- `firstprivate (list of variables)`
- `shared (list of variables)`
- `reduction (operator : list of variables)`
- `if (logical expression)`

---

All clauses behave exactly as with `parallel for` directive.
Work Sharing in Parallel Regions

Problem:

• All threads execute identical code within parallel region.
• It is not very useful to let them do exactly the same.
• Work needs to be explicitly shared among threads.

Divide work based on thread number:

```c
#pragma omp parallel private( whoami, nthreads)
{
    nthreads = omp_get_num_threads();
    whoami = omp_get_thread_num();

    do_the_work( whoami, nthreads);
}
```
Work Sharing Constructs in OpenMP

Loop scheduling with the for-directive:

```
int i;
...
#pragma omp parallel private(i)
{
#pragma omp for
    for (i=0; i<N; i++) {
        array[i] = ... ;
    }
}
```

- Work sharing / loop scheduling done implicitly by OpenMP
- Directive `parallel for` is equivalent to nested separate directives `parallel` and `for`
- Caution: Loop variables needs explicit privatization
Parallelism vs Work Sharing

The parallel directive:

- defines a parallel region
- starts team of worker threads
- performs a synchronization barrier
- terminates team of worker threads

The for directive:

- does not start / stop any threads
- distributes loop iterations among (already running) threads
- within a parallel region
- performs a synchronization barrier
Region-parallelized code:

```c
#pragma omp parallel private(i,j,x,y) 
{
    #pragma omp for
    for (i=0; i<M; i++) {
        for (j=0; j<N; j++) {
            x = (double) i / (double) M;
            y = (double) j / (double) N;
            depth[i,j] = mandelval( x, y, max);
        }
    }

    #pragma omp for
    for (i=0; i<M; i++) {
        for (j=1; j<N-1; j++) {
            dith[i,j] = 0.5 * depth[i,j] 
                        + 0.25 * depth[i,j-1] 
                        + 0.25 * depth[i,j+1];
        }
    }
}
```

Advantages:

- Larger parallel region
- Less thread termination/recreation overhead
- Less communication overhead
- Opportunity for further optimization...

Compiler Directives: OpenMP

Clemens Grelck
Computer Systems Architecture
University of Amsterdam
Region-parallelized code:

```c
#pragma omp parallel private(i,j,x,y)
{
    #pragma omp for nowait schedule(static)
    for (i=0; i<M; i++) {
        for (j=0; j<N; j++) {
            x = (double) i / (double) M;
            y = (double) j / (double) N;
            depth[i,j] = mandelval( x, y, max);
        }
    }

    #pragma omp for nowait schedule(static)
    for (i=0; i<M; i++) {
        for (j=1; j<N-1; j++) {
            dith[i,j] = 0.5 * depth[i,j]
                        + 0.25 * depth[i,j-1]
                        + 0.25 * depth[i,j+1];
        }
    }
}
```

New clause: `nowait`:

- Effect: no synchronization barrier after work sharing construct

Advantages:

- Avoid unnecessary synchronization overhead
- After first parallel loop: No need to wait for any other thread to finish due to restricted data dependence
- After second parallel loop: Synchronization done by parallel section anyways

Compiler Directives: OpenMP

Clemens Grelck
Computer Systems Architecture
University of Amsterdam
Non-Loop Work Sharing: Parallel Sections

Example:

```c
#pragma omp parallel
{
    ...
    #pragma omp sections
    {
        #pragma omp section
        {
            ...
        }
        #pragma omp section
        {
            ...
        }
        #pragma omp section
        {
            ...
        }
    }
    ...
}
```

How it works:

- Each parallel section is executed by exactly one thread.
- Threads execute different (maybe unrelated) code.
- Mapping of threads to sections is done implicitly.
- \#threads > \#sections:
  - Some threads do nothing.
- \#threads < \#sections:
  - Some threads do multiple sections.
- Synchronization barrier completes `sections` directive.
- The `parallel` directive and the `sections` directive can be combined into the `parallel sections` directive (in analogy to the `parallel for` directive).
Parallel Sections

Clauses of sections directive:

- **private** (*list of variables*)
  - private copy
  - not initialized from surrounding context

- **firstprivate** (*list of variables*)
  - private copy
  - initialized with pre-section value

- **lastprivate** (*list of variables*)
  - private copy
  - value in last section propagated outside

- **reduction** (*operator : list of variables*)
  - private copy
  - each section’s value folded with reduction operator and result propagated outside
Parallel Sections Example

Simulation Program:

- Read input data from file.
- Perform three independent preprocessing steps:
  1. Interpolation of input data to regular grid.
  2. Gathering statistics about input data.
- Perform simulation.

```c
data = read_input( );
#pragma omp parallel sections
{
    #pragma omp section
    {
        grid = interpolate( data);
    }
    #pragma omp section
    {
        stats = compute_statistics( data);
    }
    #pragma omp section
    {
        rands = generate_rand_params( );
    }
}
simulate( grid);
```
Assigning Work to a Single Thread

The single directive:

- Work sharing construct
- Exactly one thread executes code block
- Synchronization barrier afterwards
- Useful to “interrupt” parallel region
- nowait clause leaves out synchronization
- Semantically equivalent but usually more efficient than two separate parallel regions

```
#pragma omp parallel
  shared( in, out, len, array)
{
  ...
  #pragma omp single
  {
    read_array( in, array, len);
  }
  #pragma omp for
  for (i=0; i<len; i++) {
    array[i] = fun( array[i]);
  }
  #pragma omp single nowait
  {
    write_array( out, array, len);
  }
  ...
}
```
Work Sharing Constructs in OpenMP

Restrictions:

- Single point of entry.
  - No jumps into work sharing construct.

- Single point of exit.
  - No jumps out of work sharing construct.
  - No `return` or `break` statements.

- All threads must encounter work sharing construct in same order.
  - No subsets of thread teams.

- No nesting
  - Control reaches work sharing construct while already executing another one:
    * Programming error.
    * Behaviour undefined.

Compiler Directives: OpenMP

Clemens Grelck
Computer Systems Architecture
University of Amsterdam
Orphaned Work Sharing Constructs

What is that?

- Work sharing construct outside lexical scope of surrounding parallel region.
- Perfectly fine with OpenMP.
- Work sharing construct is associated with innermost parallel region in dynamic function call tree.

No parallel region around?

- If work sharing construct is encountered during serial execution, it behaves like when executed by a team of a single thread.

Example:

```c
void work() {
    ...
    #pragma omp parallel
    {
        initialize(array, N);
    } ...
}

void initialize(int *a, int N) {
    int i;
    #pragma omp for
    for (i=0; i<N; i++) {
        array[i] = ...;
    }
}
```

Compiler Directives: OpenMP

Clemens Grelck
Computer Systems Architecture
University of Amsterdam
Thread Private Global Variables

Idea:

- Global variables are always shared between all threads.
- Sometimes, it is more convenient to have individual copy for each thread.
- Analogy to thread specific data (TSD) in PThreads.
- Thread private instances of global variables survive across different parallel regions.
- Conceptually, there are N+1 instances, one for the master thread and one for each worker thread.

Example:

```c
#pragma omp threadprivate int my_id;

int main()
{
    int my_id = omp_get_thread_num();
    ...
    ...
}
```

Compiler Directives: OpenMP

Clemens Grelck
Computer Systems Architecture
University of Amsterdam
The **copyin** clause:

- Usually, there is no way to exchange data stored in thread private variables between threads.
- The **copyin** clause allows to initialize all copies with master thread’s value when entering parallel region.

### Example:

```c
#pragma omp threadprivate
int N, my_id;

int main()
{
    N = ...;

    #pragma omp parallel copyin (N)
    {
        my_id = omp_get_thread_num();
        ...
        N = fun( N, my_id);
    }
    ...
```

Compiler Directives: OpenMP

Clemens Greleek
Computer Systems Architecture
University of Amsterdam
Synchronization Barrier in OpenMP

The **barrier** directive:

- All threads wait at barrier until last thread reaches barrier.
- Allows for explicit control over synchronization barriers independent of work sharing constructs.

**Example:**

```c
#pragma omp parallel
{
    ...
    #pragma omp barrier
    ...
}
```

Compiler Directives: OpenMP

Clemens Grelck

Computer Systems Architecture

University of Amsterdam
Nested Parallel Regions

Idea:

• Parallel regions may be nested lexically or dynamically.
• Additional level of parallelism.
• New team of threads started.
• Creator thread becomes master of that new team.
• New feature of OpenMP-3.

How does it work?

• New clause: `num_threads(num)`
• Controls the team size
• Overall thread number remains constant (as before)
• `omp_get_num_threads()` and `omp_get_thread_num()` refer to innermost team
• Restriction of outer team size leaves threads for inner teams

Compiler Directives: OpenMP

Clemens Grelck
Computer Systems Architecture
University of Amsterdam
Nested Parallel Regions

Example:

```c
void schedule_tasks( )
{
    int myindex;
    int threads = omp_get_num_threads();

    #pragma omp parallel private( myindex) num_threads( threads/2);
    {
        myindex = get_next_task();
        do {
            process_task( myindex);
            myindex = get_next_task();
        } while (myindex != -1);
    }
}

void process_task( myindex)
{
    int i;

    #pragma omp parallel for num_threads( 2)
    for (i=0; i<N; i++) {
        array[myindex,i] = ... ;
    }
}
```

Compiler Directives: OpenMP

Clemens Grelck
Computer Systems Architecture
University of Amsterdam
Parallel Sections Example Reloaded

Simulation Program:

• Read input data from file.
• Perform three independent preprocessing steps:
• Perform simulation.

Use num_threads clause:

• Restrict outer team size.
• Leave threads for exploiting parallelism within sections.

data = read_input();
#pragma omp parallel sections num_threads(3) {
  #pragma omp section
  {
    grid = interpolate( data);
  }
  #pragma omp section
  {
    stats = compute_statistics( data);
  }
  #pragma omp section
  {
    rands = generate_rand_params( );
  }
}
simulate( grid);
Coarse-Grained Task Parallelism

Problem:

- In the long run data parallel loops are too inflexible

Directive: task

- Syntax: `#pragma omp task {...}
- Spawns asynchronous task for executing block
- Original thread continues with code following block
- Clauses: `if`, `private`, `shared`, `firstprivate`, etc

Directive: taskwait

- Syntax: `#pragma omp taskwait`
- Waits for all spawned tasks
- Clauses: none

Compiler Directives: OpenMP

Clemens Greilck
Computer Systems Architecture
University of Amsterdam
Task Parallelism Example

Recursive definition of Fibonacci numbers:

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

    if (n<2) {
        return n;
    }
    else {
        #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 n, f;
    printf("Enter number:");
    scanf("%d", &n);
    #pragma omp parallel shared(n,f)
    {
        #pragma omp single
        {#pragma omp single
            f = fib(n);
        }
        printf("\nfib(%d) = %d\n", n, f);
        return 0;
    }
}
```

Compiler Directives: OpenMP

Clemens Grelck
Computer Systems Architecture
University of Amsterdam
Low-level Synchronization

Similar API as PThreads:

```c
void omp_init_lock( omp_lock_t *lock);
void omp_destroy_lock(omp_lock_t *lock);
void omp_set_lock(omp_lock_t *lock);
void omp_unset_lock(omp_lock_t *lock);
int omp_test_lock(omp_lock_t *lock);
```

Nested critical sections:

- Same problems as with mutex locks in PThreads.
- May deadlock easily.
- OpenMP provides no specific means for deadlock prevention or detection.
Conclusion

Advantages:

• Simpler to use than other concepts, for example PThreads.
• OpenMP implementation responsible for most organizational aspects.
• Simplicity facilitates experimentation with alternatives, e.g. scheduling techniques.
• Large applications may be parallelized incrementally.

Disadvantages:

• Simplicity reduces programmers’ reflections on parallelism.
• Insight into impact of certain directives / clauses is reduced.
• False directives may lead to wrong — even non-deterministic — behaviour.
• False directives are easy to write but hard to find.
• Performance considerations lead to low-level “expert” code.
OpenMP Research

The future of OpenMP:

- New extensions for GPGPUs
- Heterogeneous computing
- Distributed computing
- ...

Compiler Directives: OpenMP

Clemens Grelck
Computer Systems Architecture
University of Amsterdam