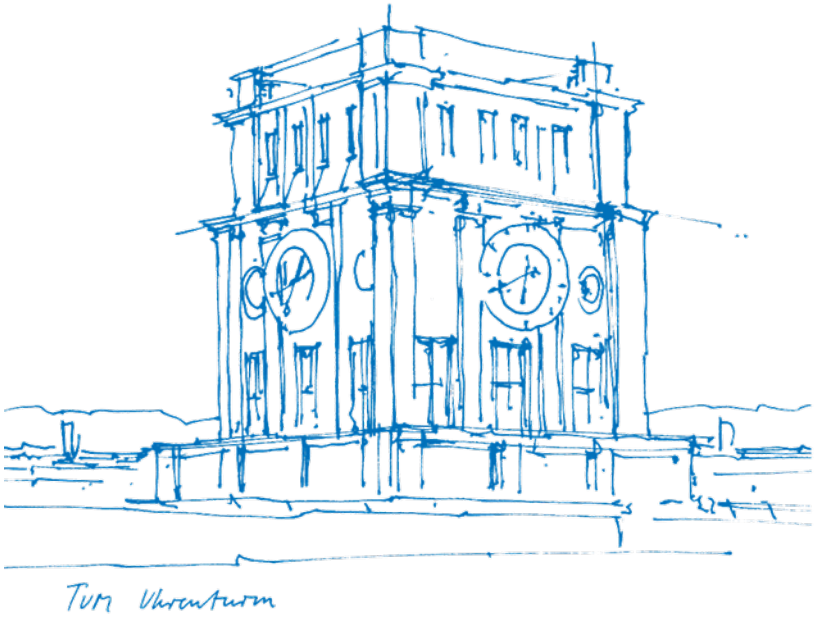
Parallel Programming Tutorial - OpenMP 1

M.Sc. Andreas Wilhelm
Technichal University Munich
May 29, 2017



Solution for Assignment 1

Solution for Assignment 2 (1/2)

The master thread...
allocates threads, arguments, and histograms
computes number of chunks before creating threads
collects the local histograms
Each worker thread...
grabs chunks in an endless loop from the buffer by...
...increasing the global counter protected by the mutex
...checking the new counter value against the chunks

```
static int gCounter;
static int gNoChunks;
static char* gBuffer;
static pthread_mutex_t gMutex =
   PTHREAD_MUTEX_INITIALIZER;

typedef struct {
   int*  histogram;
   char  dummy[64];
} thread_args;
```

```
void* compute_histogram(void *args) {
   char *chunk, current_word[20]="";
   thread_args *arg = (thread_args*)args;

   for (int c = 0, local_counter;;) {
      pthread_mutex_lock(&gMutex);
      local_counter = gCounter;
      ++gCounter;
      pthread_mutex_unlock(&gMutex);

      if (local_counter >= gNoChunks) return NULL;
      chunk = gBuffer + (CHUNKSIZE * local_counter);

      for (int i = 0; i < CHUNKSIZE; i++) {
         if(isalpha(chunk[i]) && i % CHUNKSIZE != 0) {
            current_word[c++] = chunk[i];
         } else {
            current_word[c] = '\0';
            int res = getNameIndex(current_word);
            if (res != -1) arg->histogram[res]++;
            c = 0;
         }
}}}
```

Solution for Assignment 1 (2/2)

```
void get_histogram(char *buffer, int* histogram, int num_threads) {
   pthread_t* thread = (pthread_t*) malloc(num_threads * sizeof(*thread));
   thread_args* args = (thread_args*) malloc(num_threads * sizeof(*args));
   int* local_histograms = (int*)calloc(num_threads * NALPHABET, sizeof(int));
   // initialize global memory
   gCounter = 0;
   gNoChunks = 0;
   gBuffer = buffer;
   // compute #chunks
   for (int i = 0; buffer[i] != TERMINATOR; i += CHUNKSIZE)
           gNoChunks++;
   // build histogram by num_threads threads
   for (int t=0; t < num_threads; t++) {
      args[t].histogram = local_histograms + t * NALPHABET;
      pthread_create(&thread[t], NULL, &compute_histogram, &args[t]);
   }
   // join threads and and merge histograms
   for (int t = 0; t < num_threads; t++) {
      pthread_join(thread[t], NULL);
      for (int i = 0; i < NNAMES; i++)
         histogram[i] += local_histograms[t * NALPHABET + i];
   }
}  // free allocated memory at the end
```

Organization

Organization

Assignments

- We shifted the deadline for assignment 3 to June 5th 3.45pm
- The deadline for assignment 4 is June 12th 3.45pm

Schedule

- May 29: OpenMP 1
- June 12: OpenMP 2
- June 19: Dependence Analysis
- June 26: Loop Transformations
- July 3: MPI 1
- July 10: MPI 2
- July 17: Question session

Hints for Assignment 3

Hints for Assignment 3

- Use a profiler!
- Use `std::ref()` to pass arguments by reference to a task function
- Use the launch policy `std::launch::async` when using `std::async` to explicitly spawn new threads

OpenMP

Introduction to OpenMP

- OpenMP is an API for explicit shared-memory parallelism
- Supported by most compilers (gcc, icc, msvc, clang)
- Utilizes OS threading capabilities (e.g. Pthreads)
- Fully documented in the specification (see `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`)
- Comprised of three programming layer components

1. Compiler Directives
   - Spawning parallel regions
   - Distributing loop iterations across threads
   - Synchronization
   - ...
2. Runtime Library Routines
   - Setting/Querying the number of current threads
   - Querying thread-id's and wall-clock time
   - ...
3. Environment Variables
   - Setting number of threads
   - Binding threads to processors
   - ...

Directives

Format

```
#pragma omp <directive name> <{clause, ...}>
```

- `#pragma omp`
  Required for all OpenMP C/C++ directives
- `directive name`
  A valid OpenMP directive
- `{clause, ...}`
  Optional. Clauses can be in any order
- Most OpenMP constructs apply to a structured block

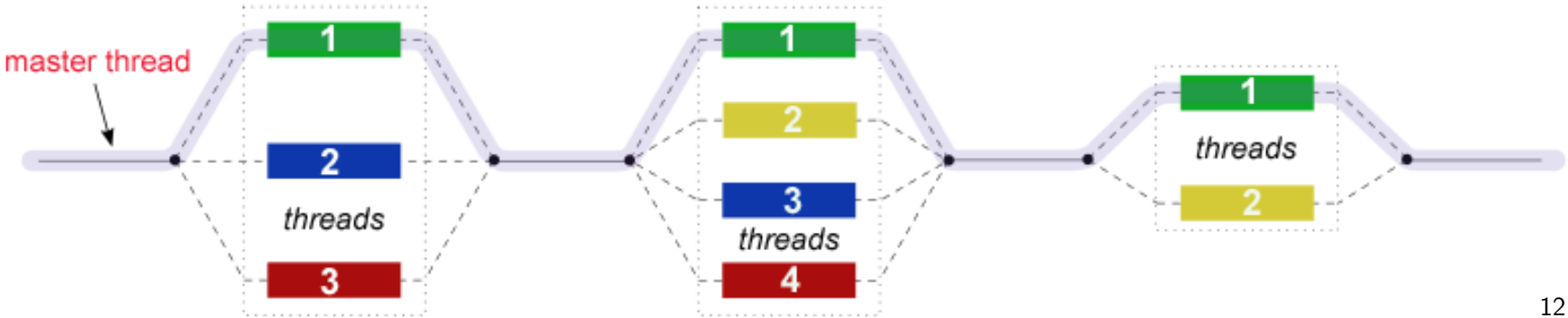
Example

```
#pragma omp parallel default(shared) private(i)
```

Parallel Region

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

- A block of code that will be executed by multiple threads
- Number of threads defined by #cpu, clauses, or env. variables
- The reaching thread (0) creates a team of N-1 threads (1, ..., N-1)
- At the end of a block there is an implicit join (barrier)
- There may be nested parallel regions



Parallel Region - Example

- `omp_get_thread_num()` returns the current thread number
- `omp_get_num_threads()` returns the number of threads

```c
#include <omp.h>
#include <stdio.h>

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

  #pragma omp parallel
  {
    printf("Hello World from thread %d\n", omp_get_thread_num());

    // only executed by main thread
    if (omp_get_thread_num() == 0)
      printf("Number of threads is %d\n", omp_get_num_threads());
  }
  return 0;
}
```

```
./hello_world
Hello World from thread 1
Hello World from thread 0
Number of threads is 3
Hello World from thread 2
```

Parallel Region - Clauses

- `if ( <scalar expression> )`
  only executed multithreaded if scalar expr. evaluates to non-zero
- `private ( <list> )`
  each thread gets a copy of variables in a comma separated list (variables might be uninitialized)
- `firstprivate/lastprivate ( <list> )`
  same as `private`, but value is copied at the entry/exit
- `shared ( <list> )`
  variables in `list` are shared (no elements of structs or arrays)
- `default ( shared | none )`
  sets the default behaviour (`none` means that data sharing needs to be explicit)
- `reduction ( <operator: list> )`
  reduction operation and associated operand
- `num_threads ( <integer expression> )`
  sets the number of threads for the parallel region
- ...

`for` Directive

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

- Worksharing construct to execute the immediately following loop by a team of threads
- Assumes that a parallel region has already been initiated (or use `omp parallel for`)
- There is an implicit barrier at the end of the loop
- Clauses:
  - `schedule ( static|dynamic|guided|runtime|auto )`
    sets the scheduling behaviour (see next slide)
  - `nowait`
    threads do not synchronize after the parallel loop
  - `ordered`
    iterations must have the same order as in a serial program
  - `collapse`
    specifies the number of (nested) loops that shall be collapsed into a larger iteration space
  - `private, firstprivate...`

`schedule` clause

- `schedule (static, chunk_size)`
  The iterations are divided into chunks of size *chunk_size* and assigned to the threads in round-robin fashion. When no chunk size is specified, the iterations are equally divided (at most one iteration per thread).
- `schedule (dynamic, chunk_size)`
  The iterations are distributed to threads in chunks as the executing threads request them. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain. Each chunk contains *chunk_size* iterations, except for the last chunk. If no chunk size is specified, it defaults to 1.
- `schdule (guided, chunk_size)`
  Similar to `dynamic`, but...
  At the beginning the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads, decreasing to 1 (or `chunk_size`, if specified).
- `schedule (auto)`
  The scheduling decision is given to the compiler/runtime system.
- `schedule (runtime)`
  The scheduling decision is deferred until run time, the schedule and chunk size are taken from internal control variables.

for Directive - Example (1/5)

```
const char *colored_digit[] = {
 "\e[1;30;1m0", "\e[1;31;1m1", "\e[1;32;1m2", "\e[1;33;1m3",
 "\e[1;34;1m4", "\e[1;35;1m5", "\e[1;36;1m6", "\e[1;37;1m7"
};
int main(int argc, char** argv) {
  unsigned int x_size = 80, y_size = 40;
  unsigned long str_len = strlen (colored_digit [0]);
  char *string_2D = (char*)malloc(x_size * y_size * str_len + y_size);

  #pragma omp parallel for schedule(runtime)
  for (unsigned long i = 0; i < y_size; i++) {
    for (unsigned int j = 0; j < x_size; j++) {
      memcpy(string_2D + ( i * x_size * str_len + i ) + (j * str_len),
             colored_digit[omp_get_thread_num()], str_len );
    }
  }
  for (unsigned int i = 0; i < y_size; i++) {
    unsigned long row = i * x_size * str_len + i ;
    printf("%s\n", string_2D + row);
  }

  printf("\033[0m");
}
```

for Directive - Example (2/5)

```
OMP_NUM_THREADS=4 OMP_SCHEDULE="STATIC" ./scheduling
```

for Directive - Example (3/5)

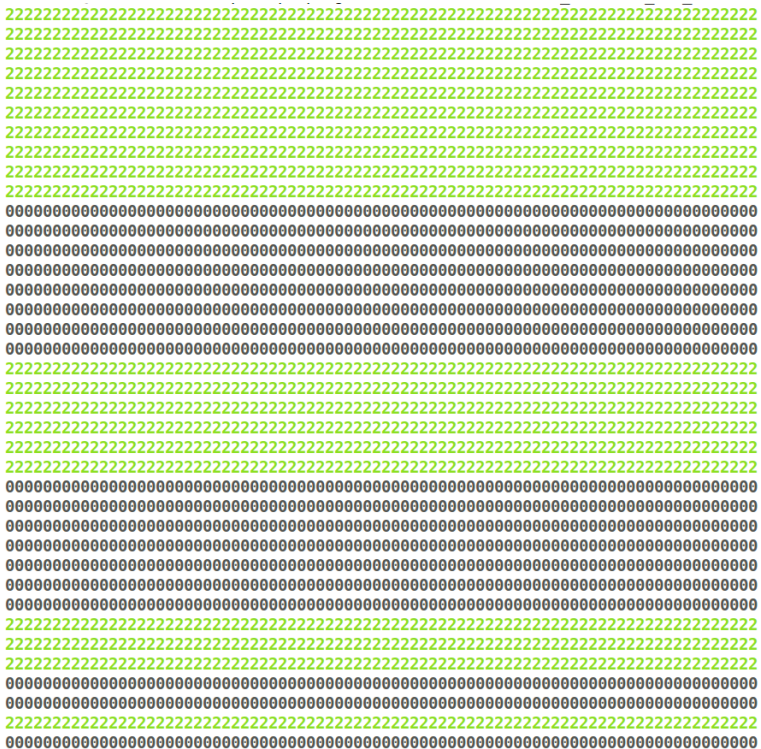
```
OMP_NUM_THREADS=4 OMP_SCHEDULE="STATIC,2" ./scheduling
```

for Directive - Example (4/5)

```
OMP_NUM_THREADS=4 OMP_SCHEDULE="DYNAMIC" ./scheduling
```

for Directive - Example (5/5)

```
OMP_NUM_THREADS=4 OMP_SCHEDULE="GUIDED" ./scheduling
```



Assignment 4

Assignment 4: Mandelbrot with OpenMP

Task

- Use OpenMP to parallelize `mandelbrot_draw()`
- Use the for-directive with appropriate clauses
- Your solution should have a speedup greater 14 on 32 cores
- Consider:
  - This time, the application is in C (no STL!)
  - Usage:
    ```
    mandelbrot_set_par -t 1 -r 720x480 -i 5000 -v [-2.0,1.0]x[-1.0,1.0] -f mandelbrot.ppm
    ```
    -t number of threads used in computation
    -i maximum number of iterations per pixel
    -r image resolution to be computed
    -v view frame that should be computed
    -p shift palette to change colors
    -f output file name
    -n no output(default: 0)

Assignment 4: Mandelbrot with OpenMP - Provided Files

- Makefile
  - contains rules to build executables
  - available targets: parallel, sequential, unit_test, checks, all (default), clean
  - 'mode=debug make [target]' to build debug version, use 'make clean' before
- main.c
  - main function
- mandelbrot_set.h
  - Header file for mandelbrot_set_*.c
- mandelbrot_set_seq.c
  - Sequential version
- student/mandelbrot_set_par.c
  - Implement the parallel version in this file