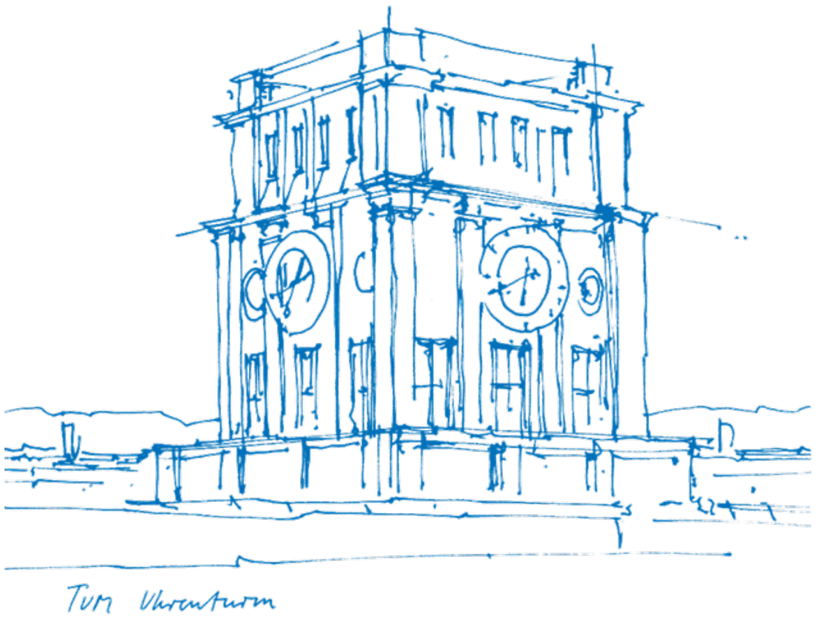
Parallel Programming Tutorial - OpenMP 1

M.Sc. Andreas Wilhelm

Technichal University Munich

May 23, 2016



Organization

Schedule

- May 23: OpenMP 1
- May 30: OpenMP 2
- June 6: Dependence Analysis
- June 13: Loop Transformations
- June 20: MPI 1
- June 27: MPI 2
- July 4: Question session

Assignment 2: Possible Solution (Speedup 3) (1/7)

1. Parallelize `playGroups()`
   - use the main thread as producer (`playGroups()`)
   - utilize N worker threads which play both group and final games
   - use a global buffer of fixed size to communicate with worker threads
   - use a global index that points to the next unprocessed buffer entry
   - use a global counter for the number of processed matches
   - the buffer contains a mode flag to distinguish group and final games

```c
#define BUFFER_SIZE 50

typedef struct {
  int mode;   // 0 for group game, 1 for final game
  int number_games;
  int group_number;
  team_t *team1;
  team_t *team2;
  team_t **successors;
} match_info;

match_info *match_buffer;
int buffer_index = -1;
int completed_games = 0;
```

Assignment 2: Possible Solution (Speedup 3) (2/7)

1. Parallelize `playGroups()`
   - wait for the consumer (condition variable `need_work`) if the buffer is full
   - fill the buffer with match information and signal that new work (condition variable (`new_work`) is available

```c
void playGroups(team_t* teams, int numWorker) {
//... some declarations
  match_buffer = malloc(BUFFER_SIZE * sizeof(match_info));

  for (g = 0; g < numWorker; ++g)
    pthread_create(threads + g, NULL, &consumer_game, NULL);

  for (g = 0; g < NUMGROUPS; ++g)
    for (i =  g * cNumTeamsPerGroup; i < (g+1) * cNumTeamsPerGroup; ++i)
      for (j = (g+1) * cNumTeamsPerGroup - 1; j > i; --j) {
        pthread_mutex_lock(&mutex);
        while (buffer_index >= BUFFER_SIZE) {
          pthread_cond_wait(&need_work, &mutex);
        }

        match_buffer[++buffer_index].group_number = g;
        match_buffer[buffer_index].team1 = teams + i;
        match_buffer[buffer_index].team2 = teams + j;
        match_buffer[buffer_index].mode = 0;

        pthread_cond_signal(&new_work);
        pthread_mutex_unlock(&mutex);
      }
//... see next slide
```

Assignment 2: Possible Solution (Speedup 3) (3/7)

1. Parallelize `playGroups()`
   - wait for the consumers to finish work after producing all matches
   - broadcast to all consumers `new_work`
   - wait for consumers to increment `completed_games` so that it is equal to `numWorker + 1`

```
  pthread_mutex_lock(&mutex);
  completed_games = 1;
  pthread_cond_broadcast(&new_work);

  for (i = 0; i < numWorker; ++i) {
    if (completed_games == numWorker+1) {
      break;
    }
    pthread_cond_wait(&work_done, &mutex);
  }
  pthread_mutex_unlock(&mutex);
}
```

Assignment 2: Possible Solution (Speedup 3) (4/7)

2. Parallelize `playFinalRound()`
   - terminate all consumer threads in case of the final match (send broadcast)
   - play final match
   - join threads

```c
void playFinalRound(int numGames, team_t** teams, team_t** successors, int numWorker) {
  pthread_mutex_lock(&mutex);
  completed_games = 0;
  pthread_mutex_unlock(&mutex);

  if (numGames == 1) {
    pthread_mutex_lock(&mutex);     // stop remaining threads
    terminate = 1;
    pthread_cond_broadcast(&new_work);
    pthread_mutex_unlock(&mutex);

// <play final match>

    for (i = 0; i < numWorker; ++i) {
      pthread_join(threads[i], NULL);
    }
  }
// continue on next slides
```

Assignment 2: Possible Solution (Speedup 3) (5/7)

2. Parallelize `playFinalRound()`
   - for all other finals: wait for consumers, fill buffers and send `new_work`

```c
void playFinalRound(int numGames, team_t** teams, team_t** successors, int numWorker) {
// ...
  else {
    for (int i = 0; i < numGames; ++i) {
      pthread_mutex_lock(&mutex);
      while (buffer_index >= BUFFER_SIZE) {
        pthread_cond_wait(&need_work, &mutex);
      }

      match_buffer[++buffer_index].number_games = numGames;
      match_buffer[buffer_index].mode = 1;
      match_buffer[buffer_index].group_number = i;
      match_buffer[buffer_index].team1 = teams[2*i];
      match_buffer[buffer_index].team2 = teams[2*i+1];
      match_buffer[buffer_index].successors = successors;

      pthread_cond_signal(&new_work);
      pthread_mutex_unlock(&mutex);
    }
// ... continue on next slide
```

Assignment 2: Possible Solution (Speedup 3) (6/7)

2. Parallelize `playFinalRound()`
   - wait for completed games to be finished

```
void playFinalRound(int numGames, team_t** teams, team_t** successors, int numWorker) {
// ...
    pthread_mutex_lock(&mutex);
    completed_games = 1;
    pthread_cond_broadcast(&new_work);
    for (i = 0; i < numWorker; ++i) {
      if (completed_games == numWorker+1) {
        break;
      }
      pthread_cond_wait(&work_done, &mutex);
    }
    pthread_mutex_unlock(&mutex);
  }
}
```

Assignment 2: Possible Solution (Speedup 3) (7/7)

3. `consumer_game()`
   - use a global flag to terminate consumer thread (`int terminate`)

```c
int terminate = 0;

void *consumer_game(void *arg) {
//...
  while (1) {
    pthread_mutex_lock(&mutex);

    while (buffer_index < 0) {
      if (terminate) { // check if consumer sent a signal to terminate
        pthread_mutex_unlock(&mutex);
        return NULL;
      }

      if (completed_games) { // increment completed games for
        ++completed_games;
        pthread_cond_signal(&work_done);
      }
      pthread_cond_wait(&new_work, &mutex);
    }

    info = match_buffer[buffer_index--]; // get next match to play
    pthread_cond_signal(&need_work);  // signal that consumer consumed a match
    pthread_mutex_unlock(&mutex);

    if (info.mode) // play final match
    else // play group match
}
```

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 applay to a structured block

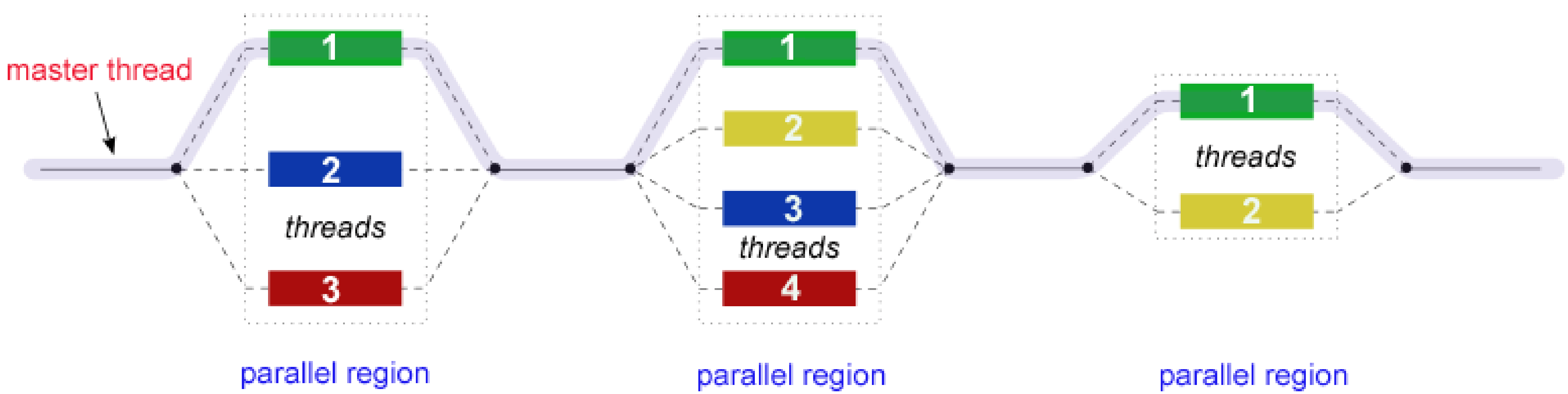
Example

```
#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 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
- 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)

```c
#include <omp.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

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;
  unsigned int 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");
  return 0;
}
```

for Directive - Example (2/5)

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

```
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
2222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
2222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
2222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
2222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
2222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
2222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
2222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
2222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
2222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
2222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
3333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333
3333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333
3333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333
3333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333
3333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333
3333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333
3333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333
3333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333
3333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333
3333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333
```

for Directive - Example (3/5)

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

```
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
22222222222222222222222222222222222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222222222222222222222222222222222
33333333333333333333333333333333333333333333333333333333333333333333333333333333
33333333333333333333333333333333333333333333333333333333333333333333333333333333
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
22222222222222222222222222222222222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222222222222222222222222222222222
33333333333333333333333333333333333333333333333333333333333333333333333333333333
33333333333333333333333333333333333333333333333333333333333333333333333333333333
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
22222222222222222222222222222222222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222222222222222222222222222222222
33333333333333333333333333333333333333333333333333333333333333333333333333333333
33333333333333333333333333333333333333333333333333333333333333333333333333333333
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
22222222222222222222222222222222222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222222222222222222222222222222222
33333333333333333333333333333333333333333333333333333333333333333333333333333333
33333333333333333333333333333333333333333333333333333333333333333333333333333333
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
22222222222222222222222222222222222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222222222222222222222222222222222
33333333333333333333333333333333333333333333333333333333333333333333333333333333
33333333333333333333333333333333333333333333333333333333333333333333333333333333
```

for Directive - Example (4/5)

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

```
00000000000000000000000000000000000000000000000000000000000000000000000000000000
11111111111111111111111111111111111111111111111111111111111111111111111111111111
00000000000000000000000000000000000000000000000000000000000000000000000000000000
11111111111111111111111111111111111111111111111111111111111111111111111111111111
00000000000000000000000000000000000000000000000000000000000000000000000000000000
11111111111111111111111111111111111111111111111111111111111111111111111111111111
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
11111111111111111111111111111111111111111111111111111111111111111111111111111111
00000000000000000000000000000000000000000000000000000000000000000000000000000000
11111111111111111111111111111111111111111111111111111111111111111111111111111111
00000000000000000000000000000000000000000000000000000000000000000000000000000000
11111111111111111111111111111111111111111111111111111111111111111111111111111111
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
11111111111111111111111111111111111111111111111111111111111111111111111111111111
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
11111111111111111111111111111111111111111111111111111111111111111111111111111111
00000000000000000000000000000000000000000000000000000000000000000000000000000000
11111111111111111111111111111111111111111111111111111111111111111111111111111111
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
11111111111111111111111111111111111111111111111111111111111111111111111111111111
00000000000000000000000000000000000000000000000000000000000000000000000000000000
11111111111111111111111111111111111111111111111111111111111111111111111111111111
00000000000000000000000000000000000000000000000000000000000000000000000000000000
11111111111111111111111111111111111111111111111111111111111111111111111111111111
00000000000000000000000000000000000000000000000000000000000000000000000000000000
11111111111111111111111111111111111111111111111111111111111111111111111111111111
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
11111111111111111111111111111111111111111111111111111111111111111111111111111111
00000000000000000000000000000000000000000000000000000000000000000000000000000000
11111111111111111111111111111111111111111111111111111111111111111111111111111111
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
11111111111111111111111111111111111111111111111111111111111111111111111111111111
00000000000000000000000000000000000000000000000000000000000000000000000000000000
```

for Directive - Example (5/5)

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

```
22222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
22222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
22222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
22222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
22222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Assignment 3: EMSim with OpenMP

Task

- Use OpenMP to parallelize the loops in `playGroups()` and `playFinalRound()` of `emsim_seq.c`
- Your solution should have a speedup greater 2.5 on 4 cores
- Consider:
  - Avoid race conditions
  - Pick proper scheduling strategy

Assignment 3: EMSim 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 - argument handling + build teams + call playEM
- emsim.h
  - Header file for emsim.c and emsim_*.c
- emsim.c
  - Defines the simulator logic
- db.h / db.c
  - Header and definition for the database accesses
- emsim_seq.c
  - Sequential version of `playGroups()` and `playFinalRound()`.
- student/emsim_par.c
  - Implement the parallel version in this file

Assignment 3: EMSim with OpenMP - Provided Files

- em.db
  - Input data: The database containing all em results.
- libsqlite3.a
  - the slite3 library to read the database
  - there is also a libsqlite3_32.a (in case you have a 32bit system) -> in that case, you have to modify the Makefile
- vis.h / vis.c
  - The visualization component
- unit_test.c
  - The unit tests that execute both the serial and parallel version to compare results.