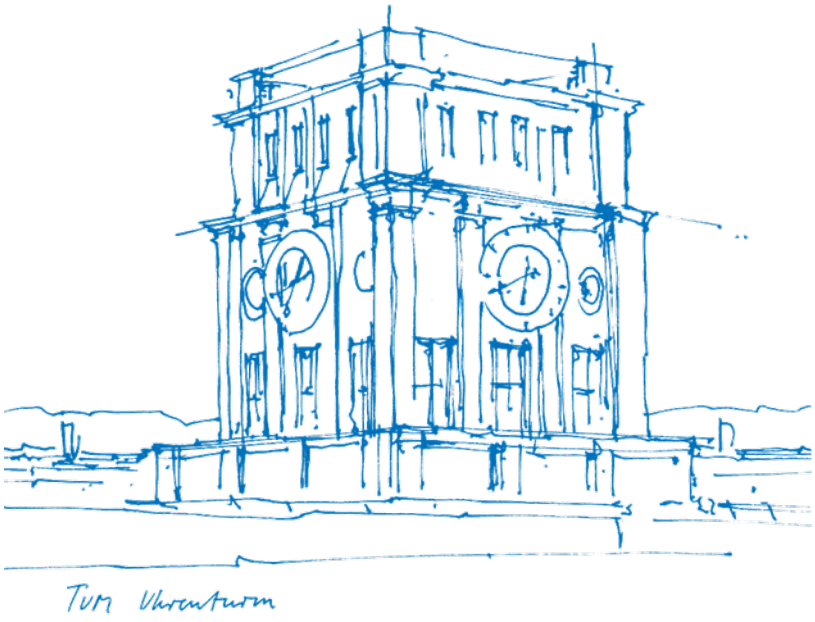
Parallel Programming Tutorial - Pthread 1

M.Sc. Andreas Wilhelm
Technichal University Munich
April 18, 2016



Organization

- Lecture could start at:
  - 4:00 PM
  - 4:15 PM
  - 4:30 PM
- Poll is available at http://doodle.com/poll/cr6udw93bfvemwgu
- Duration: as long as we need, up to 90 min

Organization

- Assignments on parallel programming techniques
- Topics
  - Pthreads (Posix Threads)
  - OpenMP (Open Multi-Processing)
  - Dependency analysis
  - MPI (Message Passing Interface)
- Code examples are in C99 (no C++)
- My email address is: andreas.wilhelm(at)in.tum.de

Assignments

- Submission of 80% of the assignments gives 0.3 bonus
- Submissions will be checked for:
  - plagiarism
  - correctness (output, threads, synchronization)
  - speedup
  - memory leaks
- Submission is done on website: https://131.159.35.135/Submission
  - requires your LRZ ID and your password
  - password is not stored and only used for authentication
  - download SSL-certificate here
- Assignment instructions are on the last slides
- Final exam will contain small programming tasks
- Solutions will be made public

Assistance on Assignments

After this week

- Given by: Helisa Dhamo and Amir Raoofy
- Email: helisa.dhamo(at)fti.edu.al / amir.raoofy(at)tum.de
- Room: 01.04.011
- Possible Date and Time:
  - Tuesday 10AM, 11AM, 12AM (60 mins.)
  - Thursday 10AM, 11AM, 12AM, 1PM, 2PM, 3PM (60 mins.)
  - Friday 10AM, 11AM, 12AM, 1PM, 2PM (60 mins.)
- Poll is available at http://doodle.com/poll/asfpt8ge6zhzdvg9

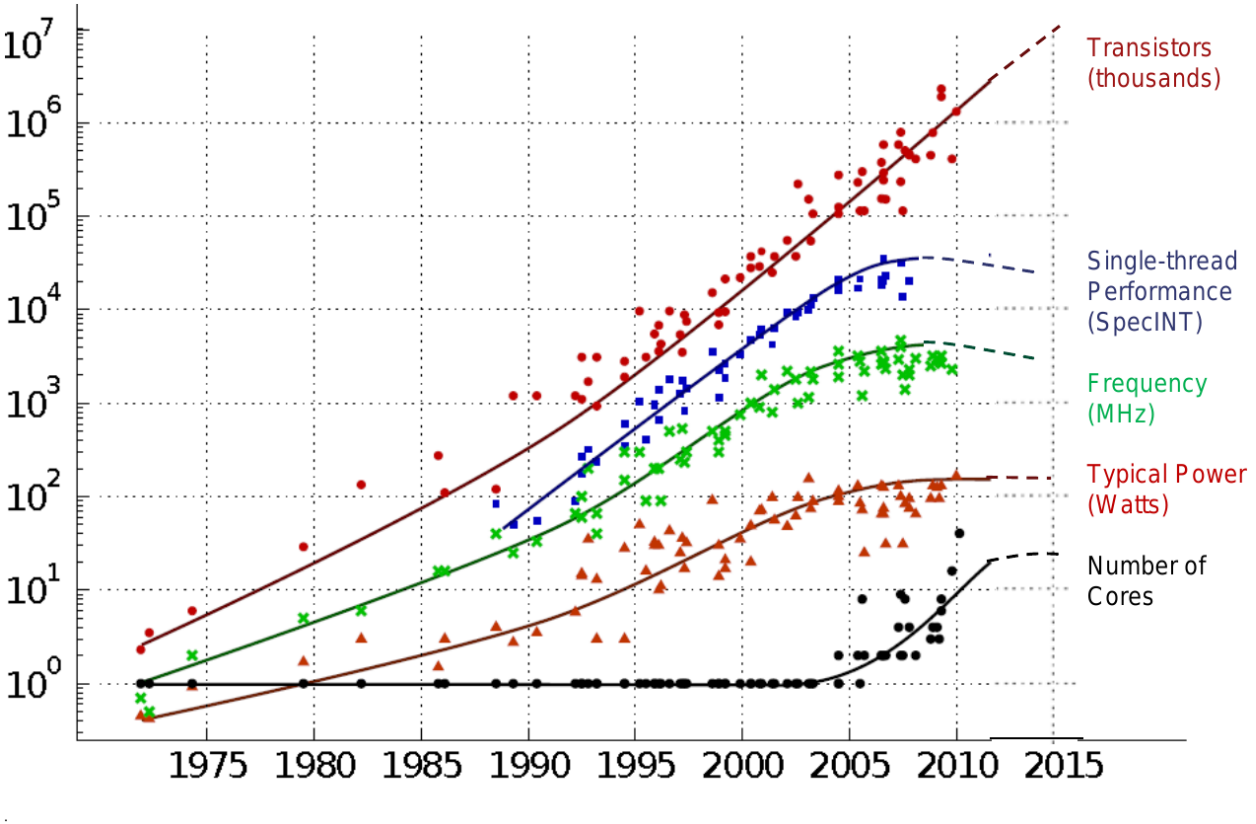
Resources

- POSIX Threads Programming
- An Introduction to Parallel Programming, by Peter Pacheco
- Programming with Posix Threads, by David Butenhof
- The Linux Programming Interface, by Michael Kerrisk
- Patterns for Parallel Programming, by Timothy G. Mattson; Beverly A. Sanders; Berna L. Massingill

Course Prerequisites

- Knowledge of C
  - memory management
  - pointers
  - global vs. static variables
- C books
  - (C89) The C Programming Language, Second Edition, by Brian W. Kernighan; Dennis M. Ritchie
  - (C99) C Primer Plus, Fifth Edition, by Stephen Prata
- Experience with Linux Command Line
- Resources
  - Book: The Linux Command Line
  - Basic video introduction: The Shell
- Knowing GCC
  - An Introduction to GCC, by Brian Gough

51th Anniversary of Moore's Law



Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

Year 2005: The Free Lunch Is Over

- A Fundamental Turn Toward Concurrency in Software
- Software doesn't get (much) faster with the next microprocessor generation
- Software developers have to rewrite their applications to use multiple processors in order top speed them up
- Parallel Programming is hard
  - to write - complex APIs and needs more code than serial version
  - to do it correctly - it's easy to introduce new bugs
  - to debug, order of thread execution is undefined
  - to make it scalable - will your applications scale with more cores?
  - better qualified developers are necessary

Posix Thread Programming

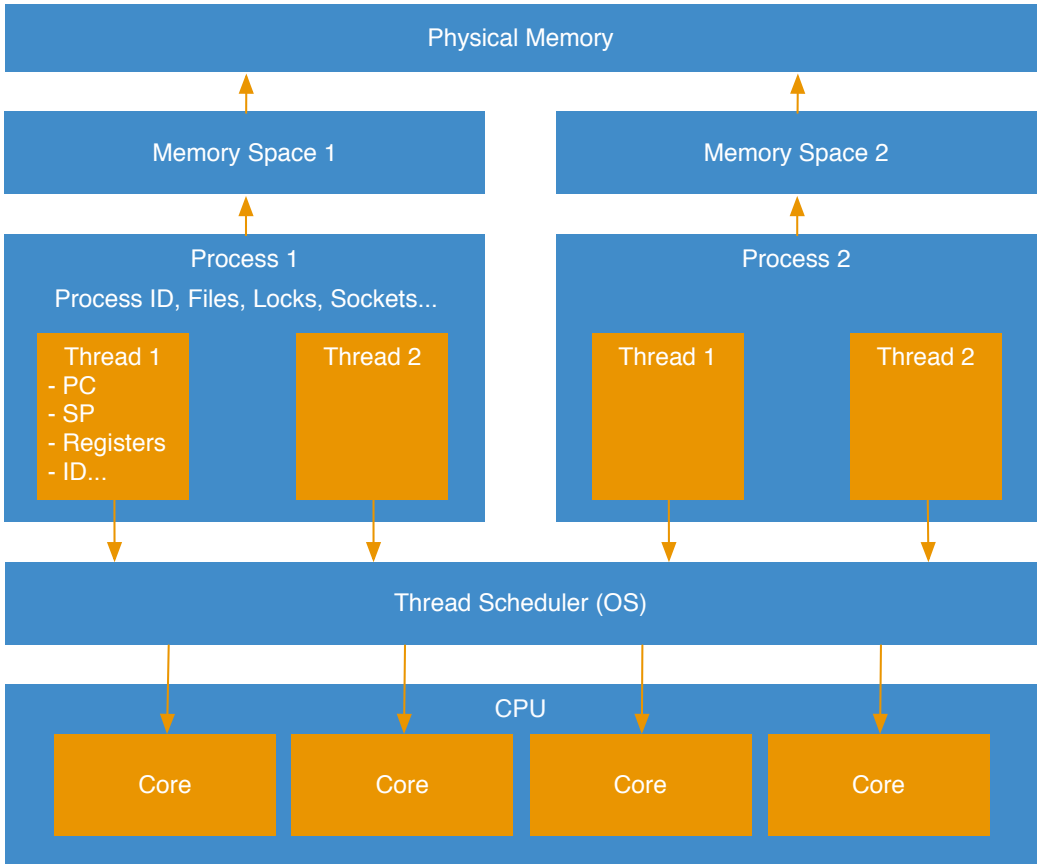
Definition: Thread

A thread is an independent stream of instructions that can be scheduled to run as such by the operating system.

POSIX Threads (Pthreads)

- Were defined in 1995 (IEEE Std 1003.1c-1995)
- Is an API that defines a set of types, functions and constants
- Is implemented with a `pthread.h` header and a thread library
- Functions can be categorized in four groups:
  - Thread management
  - Mutexes
  - Condition variables
  - Read/write locks and barriers

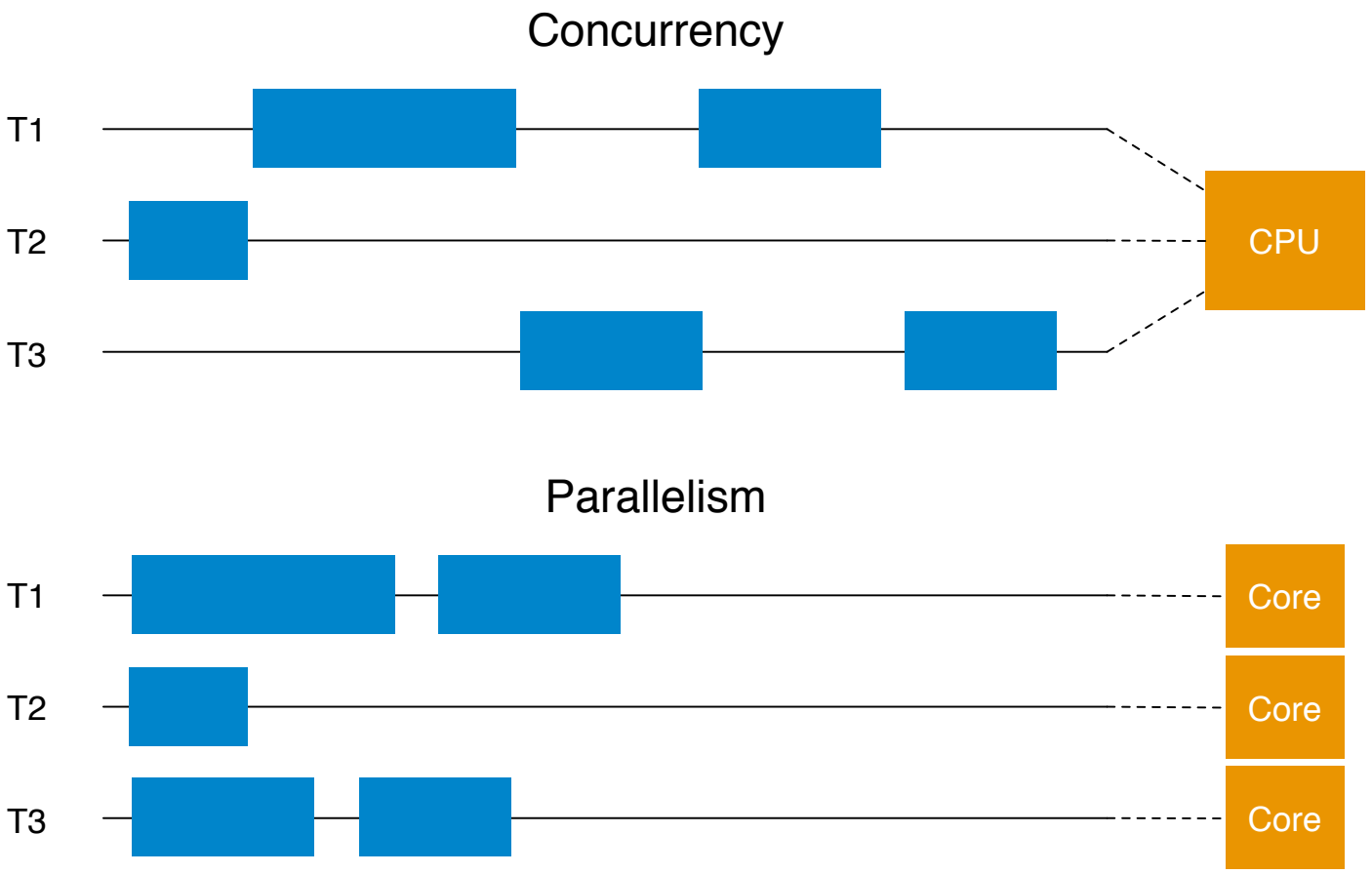
Processes vs. Threads



Why use Multithreading?

- **Performance gains**
  Parallel processing by multiple processor cores
- **Increased application throughput**
  Asynchronous system calls possible
- **Increased application responsiveness**
  Application does not need to block operations
- **Replacing process-to-process communications**
  Threads may communicate by shared-memory
- **Efficient use of system resources**
  Lightweight context switches possible
- **The ability to create well-structured programs**
  Some programs are inherently concurrent

Concurrency vs. Parallelism



Why are threads ‘faster‘ than processes?

- Creating a new process with fork() has a big overhead: whole memory must be copied
  - Waste of memory space!
- Synchronization with processes usually involves system calls.

Create Pthreads

```
1 int pthread_create(pthread_t *thread,
2                    const pthread_attr_t *attr,
3                    void *(*start_routine) (void *),
4                    void *arg);
```

- pthread_t *thread,
  - Pointer to thread identifier.
- const pthread_attr_t *attr
  - Optional pointer to pthread_attr_t to define behavior, if NULL defaults are used.
- void *(*start_routine) (void *),
  - Pointer to function prototype that is started. Function takes void pointer as argument and returns a void pointer.
- void *arg
  - Pointer to the argument that is used for the executed function.

Waiting for Pthread to finish

```
1 int pthread_join(pthread_t thread,
2                  void **retval);
```

- pthread_t thread,
  - Pointer to thread identifier, for which this function is waiting.
- void **retval
  - Optional pointer pointing to a void pointer. This can be used to return data of undefined size.

Create Pthreads - Example

```
1  #include <stdio.h>
2  #include <pthread.h>
3
4  void* hello()
5  {
6    printf("Hello World from pthread!\n");
7    return NULL;
8  }
9
10 int main(int argc, char** argv)
11 {
12   pthread_t thread;
13
14   pthread_create(&thread, NULL, &hello, NULL);
15   printf("Hello World from main!\n");
16   pthread_join(thread, NULL);
17
18   return 0;
19 }
```

Compile & Output

```
gcc --std=gnu99 -pthread -Wall
    -o hello_world hello_world.c

Hello World from main!
Hello World from pthread!
```

More than One: Hello World with Pthreads Ver. 1

```
1  #include <stdlib.h>
2
3  ...
4
5  int main(int argc, char** argv)
6  {
7    int num_threads = 3; pthread_t *thread;
8    thread = malloc(num_threads * sizeof(*thread));
9
10   for (int i = 0; i < num_threads; i++)
11     pthread_create(thread + i, NULL, &hello, NULL );
12
13   for (int i = 0; i < num_threads; i++)
14     pthread_join(thread[i], NULL );
15
16   return 0;
17 }
```

Output

```
[user]$ ./hello_world_2
Hello World from pthread!
Hello World from pthread!
Hello World from pthread!
```

Single Argument: Hello World with Pthreads Ver. 2

```
1 void * hello(void *ptr)
2 {
3   int arg = *(int*)ptr;
4   printf("Hello World from pthread %d!\n", arg);
5   return NULL;
6 }
```

Single Argument: Hello World with Pthreads Ver. 2

```
1  int main(int argc, char** argv)
2  {
3    int num_threads = 3;
4    pthread_t *thread;
5    int *arg;
6
7    thread = malloc(num_threads * sizeof(*thread));
8    arg = malloc(num_threads * sizeof(*arg));
9
10   for (int i = 0; i < num_threads; i++)
11   {
12     arg[i] = i;
13     pthread_create(thread + i, NULL, &hello, arg + i);
14   }
15
16   for (int i = 0; i < num_threads; i++)
17       pthread_join(thread[i], NULL );
18
19   free(thread);
20   free(arg);
21
22   return 0;
23 }
```

Output

```
[user]$ ./hello_world_3
Hello World from pthread 0!
Hello World from pthread 1!
Hello World from pthread 2!
```

Many Arguments: Hello World with Pthreads Ver. 3

```
1  struct pthread_args
2  {
3    long thread_id;
4    long num_threads;
5  };
6
7  void * hello(void *ptr) {
8    struct pthread_args *arg = (struct pthread_args*)ptr;
9    printf("Hello World from %ld of %ld PID = %d TID = %li!\n",
10     arg->thread_id,
11     arg->num_threads,
12     getpid(),
13     pthread_self());
14
15   return NULL;
16 }
```

Many Arguments: Hello World with Pthreads Ver. 3

```
1  #include <unistd.h>
2  int main(int argc, char** argv)
3  {
4    long num_threads = 3;
5    pthread_t *thread;
6    struct pthread_args *arg;
7    thread = malloc(num_threads * sizeof(*thread));
8    arg = malloc(num_threads * sizeof(*arg));
9
10   for (int i = 0; i < num_threads; i++)
11   {
12     arg[i].thread_id = i;
13     arg[i].num_threads = num_threads;
14     pthread_create(thread + i, NULL, &hello, arg + i);
15   }
16
17   for (int i = 0; i < num_threads; i++)
18     pthread_join(thread[i], NULL );
19   free(thread);
20   free(arg);
21   return 0;
22 }
```

Output

```
[user]$ ./hello_world_4
Hello World from pthread 1 of 3 PID = 23750 TID = 23752!
Hello World from pthread 0 of 3 PID = 23750 TID = 23751!
Hello World from pthread 2 of 3 PID = 23750 TID = 23753!
```

Return Result from Pthread in struct Argument

```
1  struct pthread_args
2  {
3    int in, out;
4  };
5
6  void * triple(void *ptr)
7  {
8    struct pthread_args *arg = ptr;
9    arg->out = 3 * arg->in;
10   return NULL;
11 }
```

Return Result from Pthread in struct Argument

```
1  int main(int argc, char** argv) {
2    int num_threads = 3; pthread_t *thread;
3    struct pthread_args *arg;
4
5    thread = malloc(num_threads * sizeof(*thread));
6    arg = malloc(num_threads * sizeof(*arg));
7
8    for (int i = 0; i < num_threads; i++){
9      arg[i].in = i;
10     pthread_create(thread + i, NULL, &triple, arg + i);
11   }
12
13   for (int i = 0; i < num_threads; i++){
14     pthread_join(thread[i], NULL );
15     printf("Triple of %d is %d\n",
16             arg[i].in,
17             arg[i].out);
18   }
19   free(thread);
20   free(arg);
21   return 0;
22 }
```

Return Result from Pthread as Pointer to Memory

```
1
2 void * triple(void *ptr) {
3
4   int *out = malloc(sizeof(*out));
5   *out = 3 * (*(int*)ptr);
6
7   return (void*)out;
8 }
```

Return Result from Pthread as Pointer to Memory

```
1  int main(int argc, char** argv) {
2    int num_threads = 3;
3    pthread_t *thread;
4    int *in;
5
6    thread = malloc(num_threads * sizeof(*thread));
7    in = malloc(num_threads * sizeof(*in));
8
9    for (int i = 0; i < num_threads; i++){
10     in[i] = i;
11     pthread_create(thread + i, NULL, &triple, in + i);
12   }
13
14   for (int i = 0; i < num_threads; i++){
15     int *out;
16     pthread_join(thread[i], (void*)&out);
17     printf("Triple of %d is %d\n", in[i], *out);
18     free(out);
19   }
20   free(thread);
21   free(in);
22   return 0;
23 }
```

What have we covered so far?

- Creating new threads with pthread_create
- Waiting for threads to finish with pthread_join
- Passing arguments to a pthread function
- Returning results from pthread function

Assignment: EMSim (EM Simulator)

Task

- EMSim simulates the European Championship 2016 by utilizing results from the last 50 years.
- Every match will be played from the group- and final-phase.
- Your task is to parallelize the function `playEM()` and `playFinalRound()` so that the application has a speedup greater 2 with 4 threads.

Assignment: EMSim - Usage

Usage of the program

- Sequential:
  `./emsim_seq <database> 1`
- Parallel:
  `./emsim_par <database> (<#threads>)`

Assignment: EMSim - playEM()

```
1  void playEM(team_t* teams) {
2    <... some declarations ...>
3
4    // play groups
5    initialize(); // necessary for the unit testing
6    for (g = 0; g < NUMGROUPS; ++g) {
7      playGroup(g, teams + (g * cTeamsPerGroup), cTeamsPerGroup,
8                successors + g * 2, successors + (numSuccessors - (g * 2) - 1),
9                bestThirds + g);
10   }
11
12   // fill best thirds
13   sortTeams(NUMGROUPS, bestThirds);
14   for (g = 0; g < numSuccessors; ++g)
15     if (successors[g] == NULL) successors[g] = bestThirds[curBestThird++];
16
17   // play final rounds
18   while (numSuccessors > 1) {
19     playFinalRound(numSuccessors / 2, successors, successors);
20     numSuccessors /= 2;
21   }
22 }
```

Assignment: EMSim - playFinalRound()

```
1  void playFinalRound(int numGames, team_t** teams, team_t** successors) {
2    team_t* team1;
3    team_t* team2;
4    int i, goals1 = 0, goals2 = 0;
5
6    for (i = 0; i < numGames; ++i) {
7      team1 = teams[i*2];
8      team2 = teams[i*2+1];
9      playFinalMatch(numGames, i, team1, team2, &goals1, &goals2);
10
11     if (goals1 > goals2)
12       successors[i] = team1;
13     else if (goals1 < goals2)
14       successors[i] = team2;
15     else {
16       playPenalty(team1, team2, &goals1, &goals2);
17       if (goals1 > goals2)
18         successors[i] = team1;
19       else
20         nsuccessors[i] = team2;
21     }
22   }
23 }
```

Assignment: EMSim - Provided Files

- Makefile
  - contains rules to build executables
  - available targets: parallel, sequential, 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 playEM().
- student/emsim_par.c
  - Implement the parallel version in this file

- 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$_t$*est.c*
  - The unit tests that execute both the serial and parallel version to compare results.

Assignment: Extract, Build, and Run

1. Extract all files to the current directory

   `tar -xvf assignment1.tar.gz`

2. Build the program

   `make [sequential] [parallel] [unit_test]`

   - sequential: build the sequential program
   - parallel: build the parallel program

- unit_test : *buildstheunittests*

3. Run the sequential program (100 repetitions)

   `student/emsim_seq em.db 1`

4. Run the parallel program (with N threads)

   `student/emsim_par em.db N`

Submission

1. Log into the webiste
2. Go to Assigments
3. Use link for Assignment 1
4. Upload your `emsim_par.c`
5. Press Submit