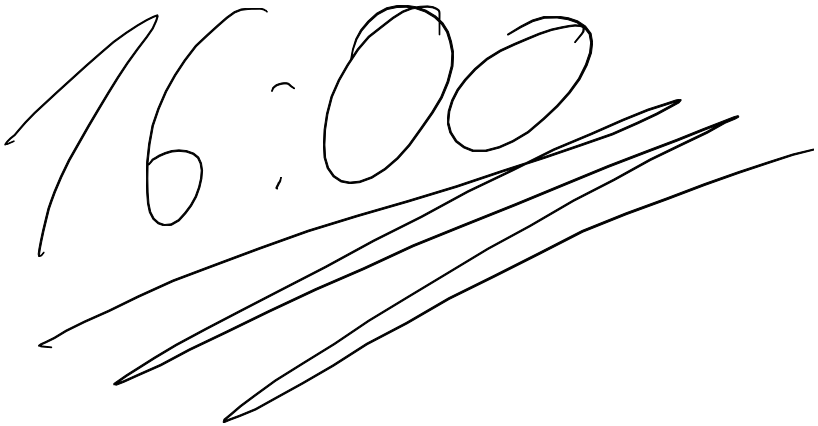
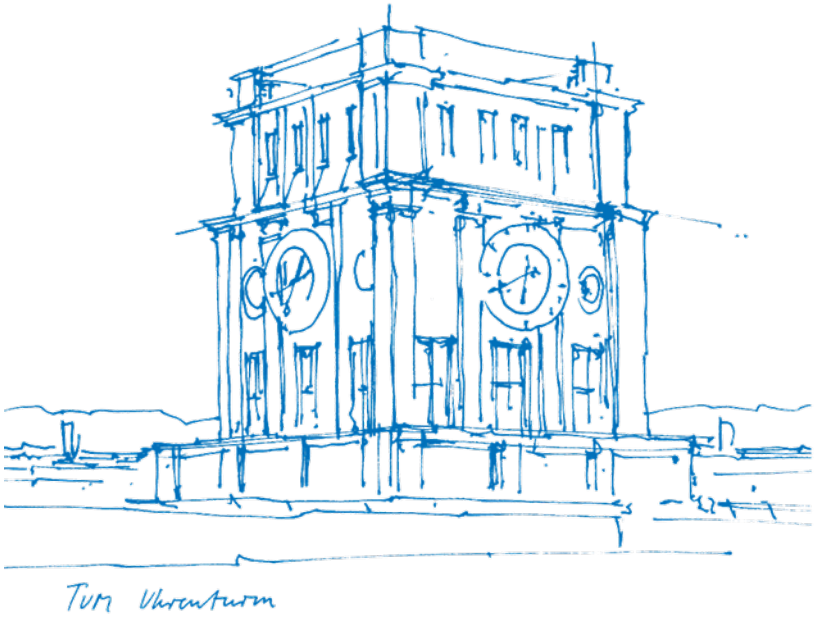
Parallel Programming Tutorial - Pthread 1

M.Sc. Andreas Wilhelm
Technichal University Munich
May 08, 2017





Organization

Organization

- Tutorial starts every Monday at 4:00 PM
- Duration: as long as we need, up to 90 min
- Assignments on parallel programming techniques
- Topics
  - Pthreads (Posix Threads)
  - C++(11/14/17)
  - OpenMP (Open Multi-Processing)
  - Dependency analysis
  - MPI (Message Passing Interface)
- Code examples are in C/C++
- My email address is: andreas.wilhelm(at)in.tum.de

Assignments

Starting this week, you have two weeks time for the first assignment!

- Submission of 80% of the assignments gives 0.3 bonus
- Submission server: https://parprogr.lrr.in.tum.de/Submission
  - requires your LRZ ID and your password
  - password is not stored and only used for authentication
- Submissions will be checked for:
  - plagiarism
  - correctness (output, threads, synchronization)
  - speedup
  - memory leaks
- Assignment instructions are on the last slides
- Final exam will contain small programming tasks (max 50% of the overall questions)
- Example solutions will be presented at the following tutorial session

Assistance on Assignments

Starting this week

- Given by: Jeeta Chacko and Jophin John
- Email: jeetachacko(at)gmail.com / jophinjohn(at)outlook.com
- Room: 01.04.011
- Date and Time:
  - Wednesday 11:30AM - 1:00PM
  - Friday 11:30AM - 1:00PM
- If you have questions, write an email to Jeeta and Jophin or visit the assistance sessions

Resources

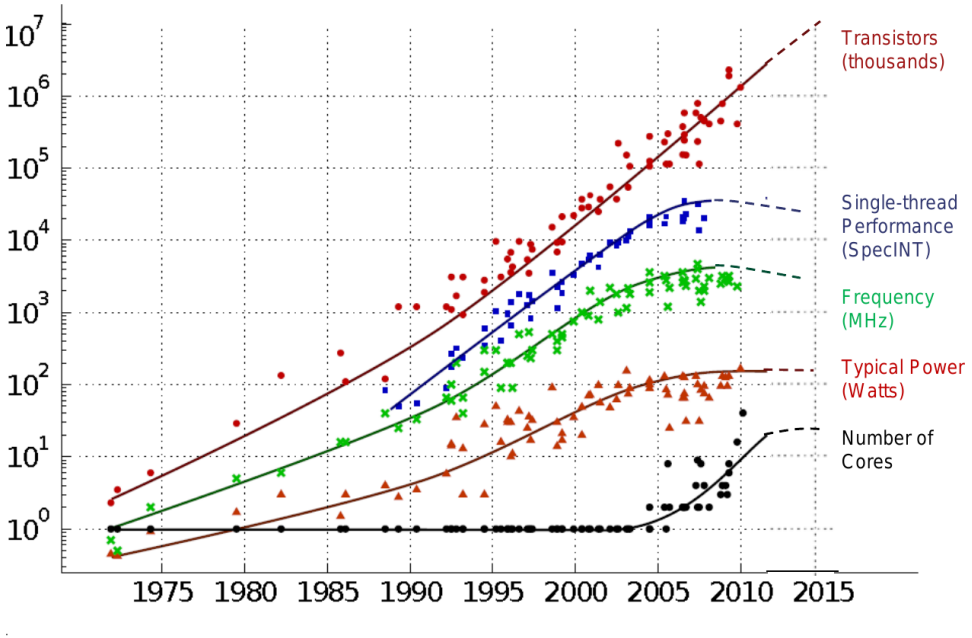
- POSIX Threads Programming
- An Introduction to Parallel Programming, by Peter Pacheco
- Programming with Posix Threads, by David Butenhof
- Patterns for Parallel Programming, by Timothy G. Mattson; Beverly A. Sanders; Berna L. Massingill
- Multithreading in Modern C++, by Rainer Grimm

Course Prerequisites

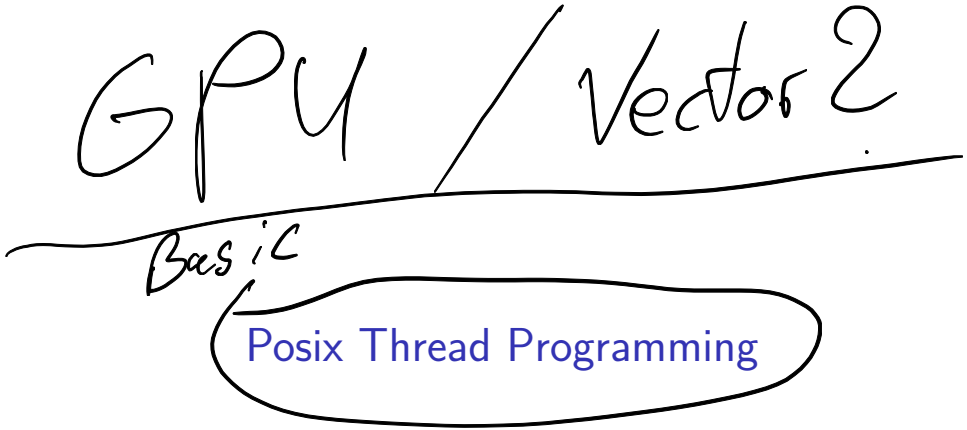
- Knowledge of C/C++
  - memory management
  - pointers /references
  - global vs. static variables
- C/C++(11/14/17) books
  - (C89) The C Programming Language, Second Edition, by Brian W. Kernighan; Dennis M. Ritchie
  - (C99) C Primer Plus, Fifth Edition, by Stephen Prata
  - (C++11/14) The C++ Programming Language, Fourth Edition, by Bjarne Stroustrup
- 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

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
- Developers have to rewrite their software so that multiple computation units are used
- Parallel Programming is hard
  - to write - higher code complexity
  - to do it correctly - easy to introduce bugs
  - to debug - order of thread execution is undefined
  - to make it scalable - will your applications scale with additional cores?
- → Qualified developers are necessary



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


GPU / Vector 2
Basic
Posix Thread Programming

Posix Thread Programming

Definition: Thread

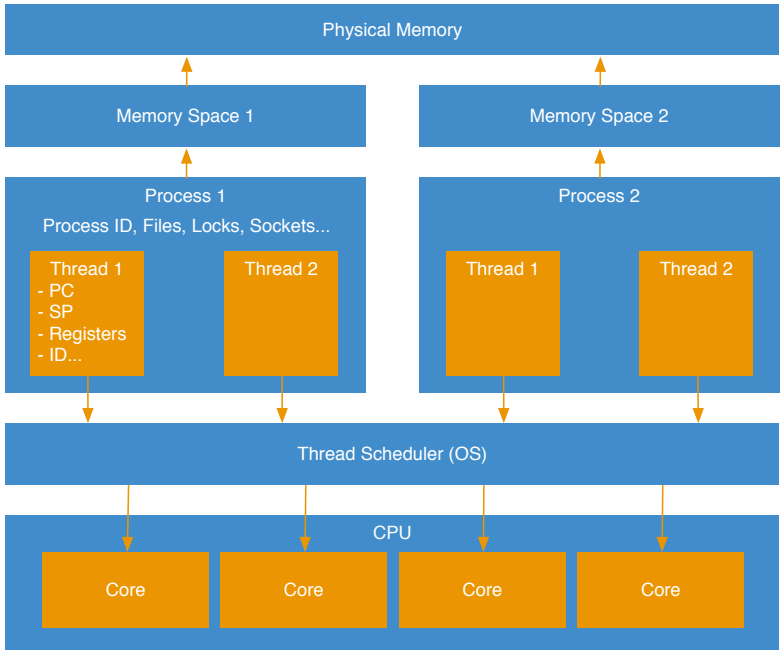
A thread is an independent stream of instructions that can be scheduled to run as such by the operating system.

POSIX Threads (Pthreads)

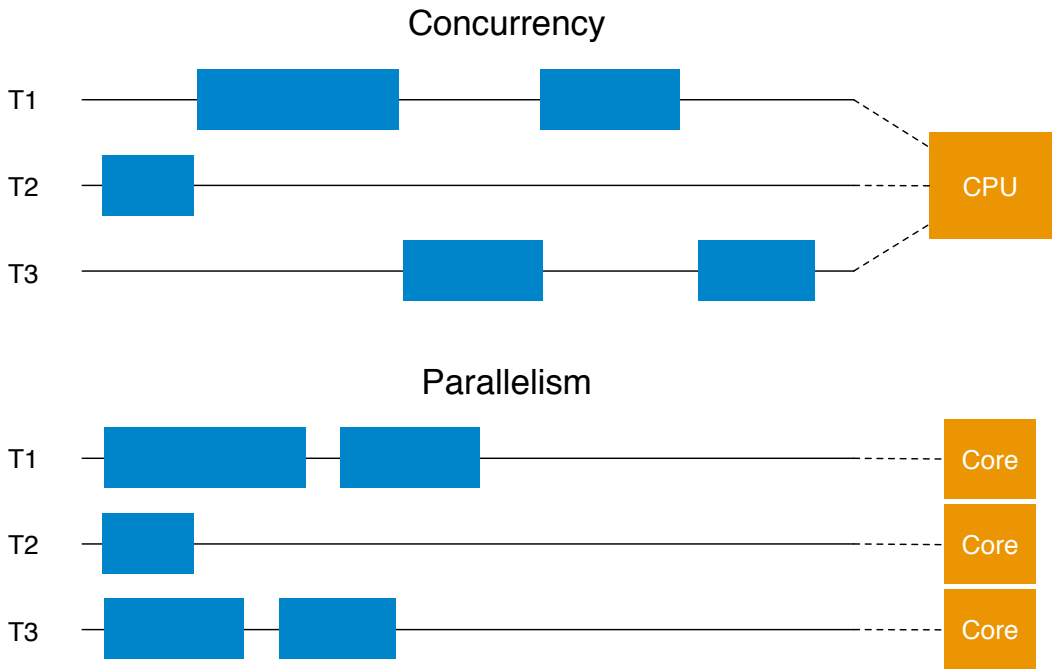
- Were defined in 1995 (IEEE Std 1003.1c-1995)
- Is an API that defines a set of types, functions and constants
- Is implemented with a `pthread.h` header and a thread library
- Natively supported by FreeBSD, NetBSD, OpenBSD, Linux, Mac OS X, Android and Solaris
- Functions can be categorized in four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

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
- **Separation of concerns**
  Some problems are inherently concurrent



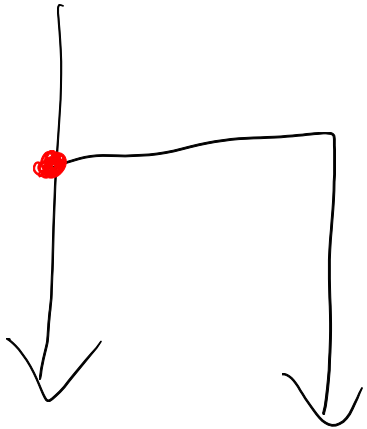
Concurrency vs. Parallelism



Pthread Syntax / Semantics

Create Pthreads

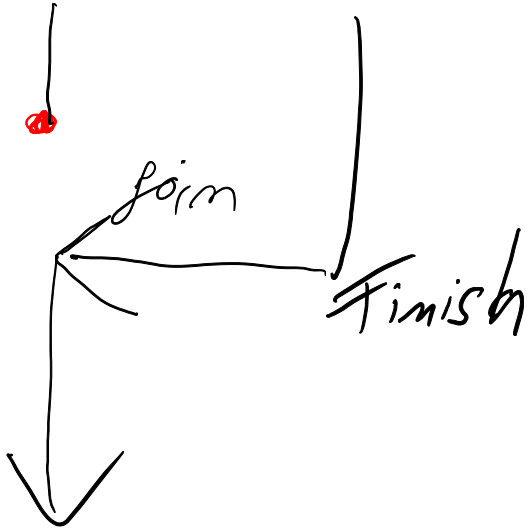
```
1 int pthread_create(pthread_t *thread,
2                    const pthread_attr_t *attr,
3                    void *(*start_routine) (void *),
4                    void *arg);
```



- 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(void* args)
5  {
6    printf("Hello World from pthread!\n");
7    return NULL;
8  }
9
10 int main()
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 }
```

Compile & Output

```
gcc -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  int main()
4  {
5    int num_threads = 3;
6    pthread_t *thread = (pthread_t*) malloc(num_threads * sizeof(*thread));
7
8    for (int i = 0; i < num_threads; i++)
9      pthread_create(thread + i, NULL, &hello, NULL );
10
11   for (int i = 0; i < num_threads; i++)
12     pthread_join(thread[i], NULL );
13 }
```

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

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() {
3    long num_threads = 3;
4    pthread_t *thread;
5    struct pthread_args *arg;
6    thread = (pthread_t*) malloc(num_threads * sizeof(*thread));
7    arg = (struct pthread_args*) malloc(num_threads * sizeof(*arg));
8
9    for (int i = 0; i < num_threads; i++) {
10     arg[i].thread_id = i;
11     arg[i].num_threads = num_threads;
12     pthread_create(thread + i, NULL, &hello, arg + i);
13   }
14
15   for (int i = 0; i < num_threads; i++)
16     pthread_join(thread[i], NULL );
17
18   free(thread);
19   free(arg)
20 }
```

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() {
2    int num_threads = 3; pthread_t *thread;
3    struct pthread_args *arg;
4
5    thread = (pthread_t*) malloc(num_threads * sizeof(*thread));
6    arg = (struct pthhread_args*) 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 Result from Pthread as Pointer to Memory

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

What have we covered so far?

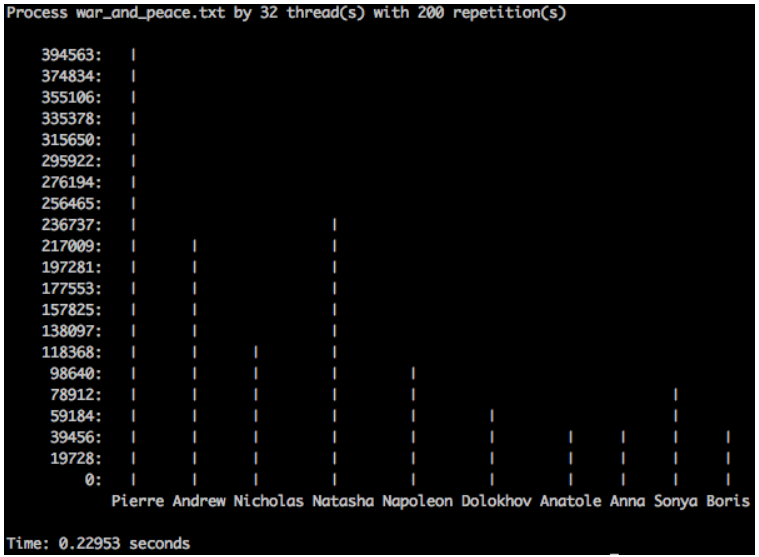
- Creating new threads with pthread_create
- Waiting for threads to finish with pthread_join
- Passing arguments to a pthread function
- Returning results from pthread function

Assignment 1: Actors in “War and Peace”

Assignment: histogram

Starting this week, you have two weeks time. Submission Server is not yet ready, will be announced in Moodle!

- `histogram` counts the occurences of ten famous actors in the book "War and Peace"
- Therefore, `get_histogram` iterates over all words and compares each word with the actor names
- The task is to parallelize `get_histogram`



Assignment: histogram

Usage of the program

- Sequential:
  `./histogram_seq war_and_peace.txt <#threads> <#repetitions>`
- Parallel:
  `./histogram_par war_and_peace.txt <#threads> <#repetitions>`

Assignment: histogram - get_histogram()

```
1  void get_histogram(int nBlocks, block_t *block, int* hist, int num_threads)
2  {
3     char current_word[20] = "";
4     int c = 0;
5
6     for (int i = 0; i < nBlocks; i++) {  // loop over all blocks
7        for (int j=0; j < BLOCKSIZE; j++) {  // loop over all characters
8
9           if(isalpha(blocks[i][j])){  //  add character to current word
10             current_word[c++] = blocks[i][j];
11          } else {  // the end of the current word
12             current_word[c] = '\0';
13             for(int k = 0; k < NNAMES; k++) {
14                if(!strcmp(current_word, names[k]))
15                   hist[k]++;
16             }
17             c = 0;
18          }
19       }
20    }
21 }
```

Assignment: histogram - 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 + file handling + call `get_histogram()`
- histogram.h
  - Header file for histogram.c and histogram_*.c
- histogram.c
  - Defines helper functions
- histogram_seq.c
  - Sequential version of `get_histogram()`.
- student/histogram_par.c
  - Implement the parallel version in this file

Assignment: histogram - Provided Files

- war_and_peace.txt
  - Input data: The book war and peace.
- unit_test.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: builds the unit tests

3. Run the sequential program (100 repetitions)

   ```
   student/histogram_seq war_and_peace.txt 1 100
   ```

4. Run the parallel program (with N threads and 100 repetitions)

   ```
   student/histogram_par war_and_peace.txt N 100
   ```

Submission

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