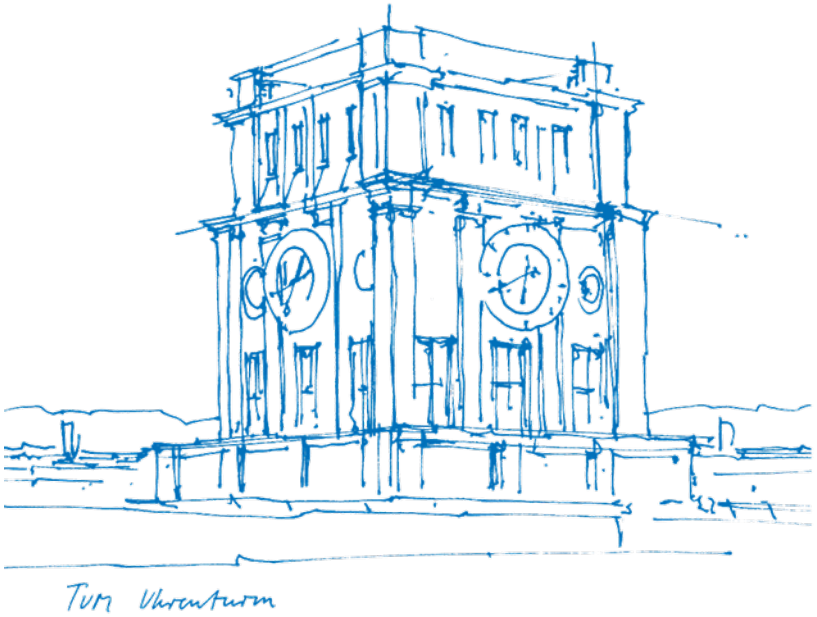
Parallel Programming Tutorial - Introduction to Pthread API

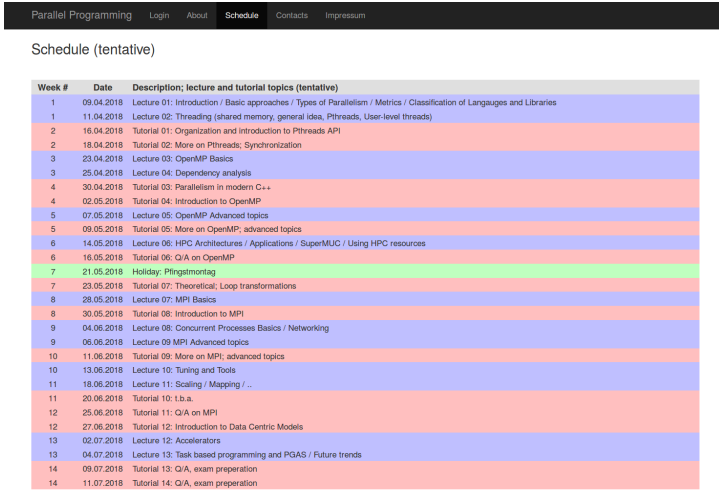
Amir Raoofy, M.Sc.
Chair for Computer Architecture and Parallel Systems (CAPS)
Technichal University Munich
16. April 2018



Organization

Organization

- Course web-page
  - parprog.lrr.in.tum.de
  - https problem is resolved!
  - Register and login using your @mytum IDs.
  - Course schedule; lecture and tutorial
  - Lecture material and tutorial slides
  - Exercises and assignment submission
- Tutorials: Wednesdays at 8:15 to 9:45
  - Always check the schedule in the web-page
- Where to find us?
  - Chair for Computer Architecture and Parallel Systems (Prof. Dr. M. Schulz)
  - My email address is: amir.raoofy(at)tum.de
  - Room: MI, 01:04:39



Week #	Date	Description; lecture and tutorial topics (tentative)
1	09.04.2018	Lecture 01: Introduction / Basic approaches / Types of Parallelism / Metrics / Classification of Langauges and Libraries
1	11.04.2018	Lecture 02: Threading (shared memory, general idea, Pthreads, User-level threads)
2	16.04.2018	Tutorial 01: Organization and introduction to Pthreads API
2	18.04.2018	Tutorial 02: More on Pthreads; Synchronization
3	23.04.2018	Lecture 03: OpenMP Basics
3	25.04.2018	Lecture 04: Dependency analysis
4	30.04.2018	Tutorial 03: Parallelism in modern C++
4	02.05.2018	Tutorial 04: Introduction to OpenMP
5	07.05.2018	Lecture 05: OpenMP Advanced topics
5	09.05.2018	Tutorial 05: More on OpenMP; advanced topics
6	14.05.2018	Lecture 06: HPC Architectures / Applications / SuperMUC / Using HPC resources
6	16.05.2018	Tutorial 06: Q/A on OpenMP
7	21.05.2018	Holiday: Pfingstmontag
7	23.05.2018	Tutorial 07: Theoretical; Loop transformations
8	28.05.2018	Lecture 07: MPI Basics
8	30.05.2018	Tutorial 08: Introduction to MPI
9	04.06.2018	Lecture 08: Concurrent Processes Basics / Networking
9	06.06.2018	Lecture 09 MPI Advanced topics
10	11.06.2018	Tutorial 09: More on MPI; advanced topics
10	13.06.2018	Lecture 10: Tuning and Tools
11	18.06.2018	Lecture 11: Scaling / Mapping / ..
11	20.06.2018	Tutorial 10: t.b.a.
12	25.06.2018	Tutorial 11: Q/A on MPI
12	27.06.2018	Tutorial 12: Introduction to Data Centric Models
13	02.07.2018	Lecture 12: Accelerators
13	04.07.2018	Lecture 13: Task based programming and PGAS / Future trends
14	09.07.2018	Tutorial 13: Q/A, exam preperation
14	11.07.2018	Tutorial 14: Q/A, exam preperation

Assignments

We release two assignments this week and you will have 2 weeks time for completing them!

- We will work on 11 assignments on parallel programming techniques
- Submission of 80% of the assignments brings you 0.3 bonus
- Submission server: https://parprog.lrr.in.tum.de/Submission/assignments
  - I will walk you through the submission work-flow at the end of todays tutorial session
- Submissions will be checked for:
  - Plagiarism, correctness (output, threads, synchronization), speedup, memory leaks
- Example solutions will be presented at the following tutorial session
- Topics
  - Pthreads (Posix Threads)
  - C++(11/14/17)
  - OpenMP (Open Multi-Processing)
  - Dependency analysis
  - MPI (Message Passing Interface)

Assistance on Assignments

Starting this week

- Given by: V.A. Suma and C. Demirsoy, R. Singh
- Emails:
  - vishnu.anilkumar-suma(at)tum.de
  - canberk.demirsoy(at)tum.de
  - rakesh.singh@tum.de
- Room: 01.06.020
- Date and Time:
  - Tuesday 14:00 - 16:00
  - Tuesday 16:00 - 18:00
- If you have questions, write an email to Vishnu and Rakesh and Canberk 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

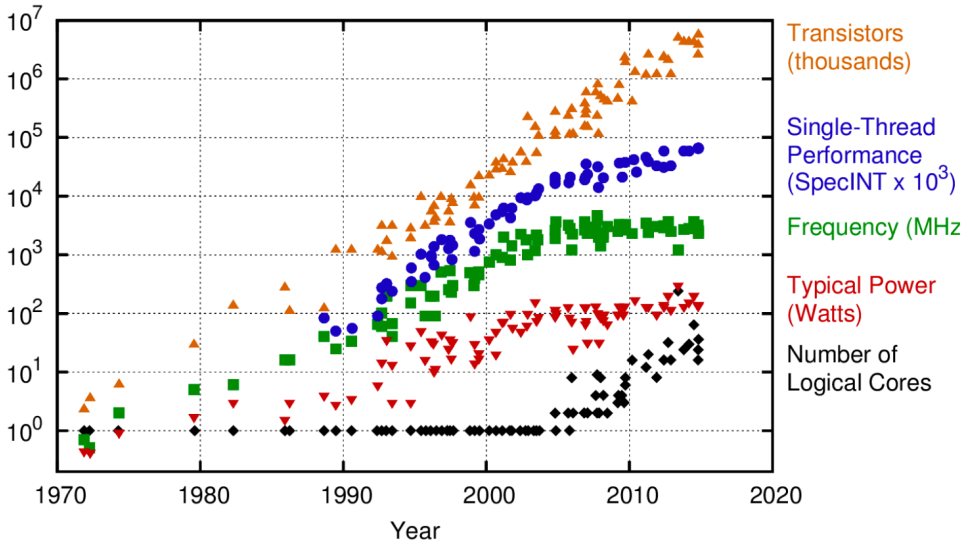
- Many thanks to my colleague Andreas Wilhelm
  - Development of the slides and tutorial material

Course Prerequisites

- knowledge of C/C++ (our code examples and assignments are all in 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 CLI
  - Book: The Linux Command Line
  - Basic video introduction: The Shell
- knowing compilers/toolchain (e.g., GCC is sufficient here)
  - 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 → YOU



Posix Thread Programming

Posix Thread Programming

Definition: (Software) Thread

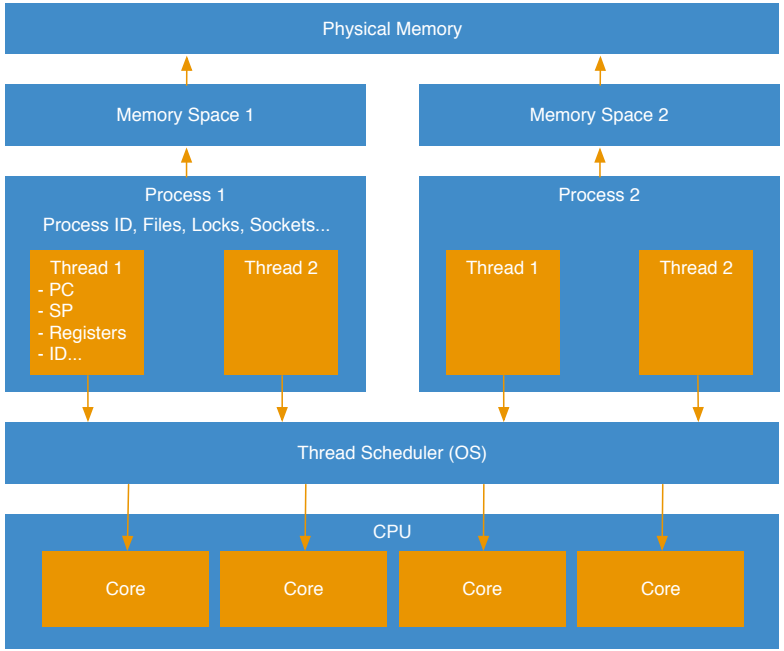
A thread is an independent stream of instructions that can be scheduled to run as such by the operating system. (Own PC and SP)

POSIX Threads (Pthreads)

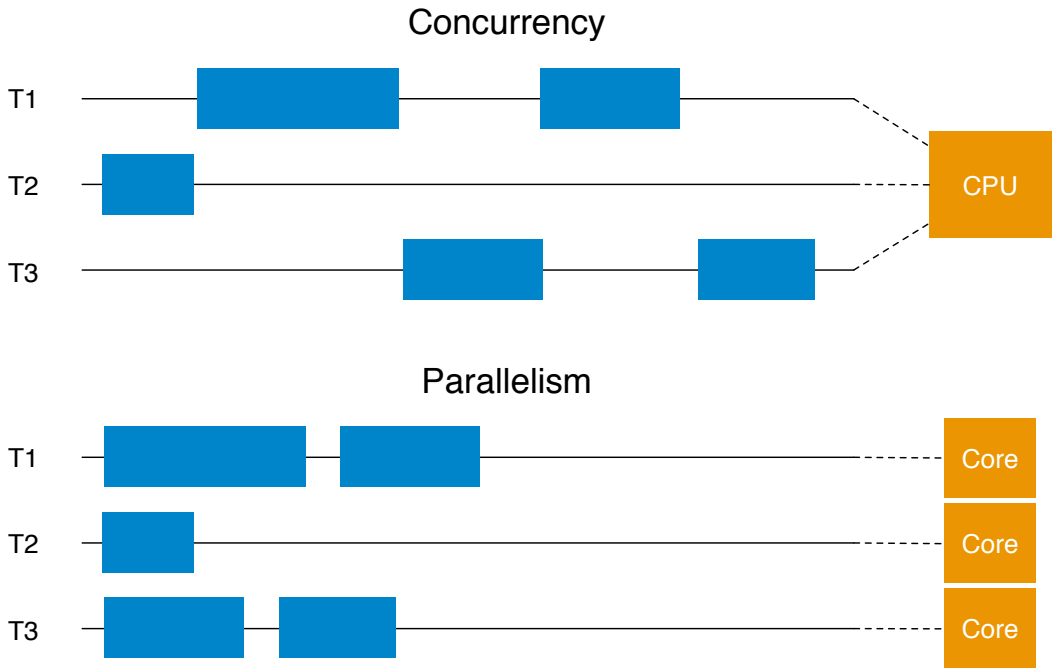
- Were defined in 1995 (IEEE Std 1003.1c-1995)
- Is an API that defines a set of types, functions and constants
- Is implemented with a `pthread.h` header and a thread library
- Natively supported by FreeBSD, NetBSD, OpenBSD, Linux, Mac OS X, Android and Solaris
- Functions can be categorized in four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

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,
  - 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.

Example 1; creating a thread

```
1  #include <stdio.h>
2  #include <pthread.h>
3
4  // function to be executed by the thread
5  void* kernel (void* args){
6    printf("hello from the thread!\n");
7    return NULL;
8  }
9
10 int main(int argc, char *argv[])
11 {
12   pthread_t thread;                              // allocate a thread
13   pthread_create(&thread, NULL, kernel , NULL);  // create the thread and start executing kernel in parallel
14   printf("hello from main\n");
15   pthread_join(thread,NULL);                     //wait for the thread to finish executing kernel
16
17   return 0;
18 }
```

Compile & Output

```
gcc -pthread -Wall -o ex1 ex1.c
./ex1

Hello from main!
Hello from the thread!
```

Example 2; creating multiple threads

```
1  int main(int argc, char *argv[])
2  {
3    //allocate the threads
4    int num_threads=4;
5    pthread_t *threads = (pthread_t*) malloc (num_threads *sizeof(pthread_t));
6
7    //create threads, start executing kernel in parallel
8    for (int i = 0; i < num_threads; ++i) {
9      pthread_create(&threads[i], NULL, kernel, NULL);
10   }
11
12   //wait for all the threads to finish executing kernel
13   for (int i = 0; i < num_threads; ++i) {
14     pthread_join(threads[i], NULL);
15   }
16
17   free(threads);
18   return 0;
19 }
```

Output

```
./ex2

Hello from the thread!
Hello from the thread!
Hello from the thread!
Hello from the thread!
```

Example 3, passing an argument to threads

```
1 void* kernel (void* args){
2   int id = *(int*)args;
3   printf("Hello from the thread, myid: %d!\n", id);
4   return NULL;
5 }
```

Example 3, passing an argument to threads (cont.)

```
1  int main(int argc, char *argv[])
2  {
3    //allocate the threads
4    int num_threads=4;
5    pthread_t *threads = (pthread_t*) malloc (num_threads*sizeof(pthread_t));
6    int* id = (int*) malloc (num_threads*sizeof(int));
7
8    //create threads, start executing kernel in parallel
9    for (int i = 0; i < num_threads; ++i) {
10     id[i]=i; //set the id for the threads
11     pthread_create(&threads[i], NULL, kernel, id+i); //pass the id as argument to the threads
12   }
13
14   //wait for all the threads to finish executing kernel
15   for (int i = 0; i < num_threads; ++i) {
16     pthread_join(threads[i], NULL);
17   }
18
19   free(threads); free(id);
20   return 0;
21 }
```

Output

```
./ex3

Hello from the thread, myid: 1!
Hello from the thread, myid: 0!
Hello from the thread, myid: 2!
Hello from the thread, myid: 3!
```

Example 4; process and thread IDs

```
1 void* kernel (void* args){
2   int id = *(int*)args;
3   printf("Hello from the thread, myid: %d, PID: %d, TID:%d!\n", id, getpid(), (int) gettid());
4   return NULL;
5 }
```

Output

```
./ex4

Hello from the thread, myid: 1, PID: 12347, TID:12349!
Hello from the thread, myid: 0, PID: 12347, TID:12348!
Hello from the thread, myid: 2, PID: 12347, TID:12350!
Hello from the thread, myid: 3, PID: 12347, TID:12351!
```

Example 5, passing multiple arguments

```
1  struct pthread_args
2  {
3    long thread_id ;
4    long num_threads ;
5  };
6
7  void* kernel (void* args){
8    struct pthread_args *arg = (struct pthread_args*) args;
9    printf("Hello from the thread, number of threads: %ld, myid: %ld, PID: %d, TID:%d!\n", \
10   arg->num_threads, arg->thread_id, getpid(), (int) gettid());
11   return NULL;
12 }
```

Example 5, passing multiple arguments (cont.)

```
1  int main(int argc, char *argv[])
2  {
3    int num_threads=4;
4    pthread_t *threads = (pthread_t*) malloc (num_threads*sizeof(pthread_t));
5    struct pthread_args* args = (struct pthread_args*) malloc (num_threads*sizeof (struct pthread_args));
6
7    for (int i = 0; i < num_threads; ++i) {
8      //set the id and num threads in args for the threads
9      args[i].thread_id=i;
10     args[i].num_threads=num_threads;
11     //pass the args as argument to the threads
12     pthread_create(&threads[i], NULL, kernel, args+i); // passing args[i] to threads[i]
13   }
14
15   for (int i = 0; i < num_threads; ++i) {
16     pthread_join(threads[i], NULL);
17   }
18
19   free(threads); free(args);
20   return 0;
21 }
```

Example 6, how to get data out of threads

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

Example 6, how to get data out of threads (cont.)

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

Example 7, return data from threads

```
1 void* kernel_double (void* args){
2   int in = *(int*) args;
3   int *out = (int*) malloc (1*sizeof (int));
4   *out = 2*in;
5   return (void*)out;
6 }
```

Example 7, return data from threads (cont.)

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

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

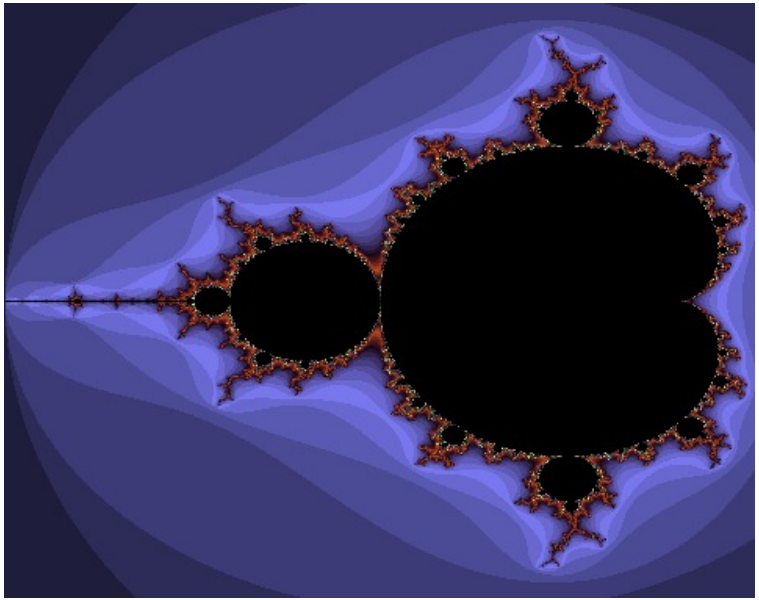
Assignment 1: “Mandelbrot set” in parallel

Assignment: Mandelbrot

Starting this week, you have two weeks time.

- Use Pthreads to parallelize mandelbrot_draw()
- Your solution should have a speedup greater than 3.0 using 4 threads

```
1  void mandelbrot_draw( ... some args ) {
2    ...
3    for (int i = 0; i < y_resolution; i++)
4    {
5      for (int j = 0; j < x_resolution; j++)
6      {
7        //embarrassingly parrallel calculation of pixels
8        ...
9      }
10   }
11 }
```



Assignment: Mandelbrot (cont.)

Build the program

- Makefile:
  `make`

Usage of the program

- Sequential:
  `./mandelbrot_set_seq -h`
- Parallel:
  `./mandelbrot_set_par -t 4 -r 480x380 -i 1000 -v [-2.0,0.5]x[-1.25,1.25] -f mandelbrot.ppm`

Assignment: Mandelbrot - 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 `draw_mandelbrot()`
- mandelbrot_set.h
  - Header file for mandelbrot_set_*.c
- mandelbrot.c
  - Defines helper functions
- mandelbrot_set_seq.c
  - Sequential version of `draw_mandelbrot()`
- student/mandelbrot_set_par.c
  - Implement the parallel version in this file

Assignment: Mandelbrot - provided files (cont.)

- 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 (with default parameters)

   ```
   student/mandelbrot_set_seq
   ```

4. Run the parallel program (with 4 threads)

   ```
   student/mandelbrot_set_par -t 4
   ```

Submission

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