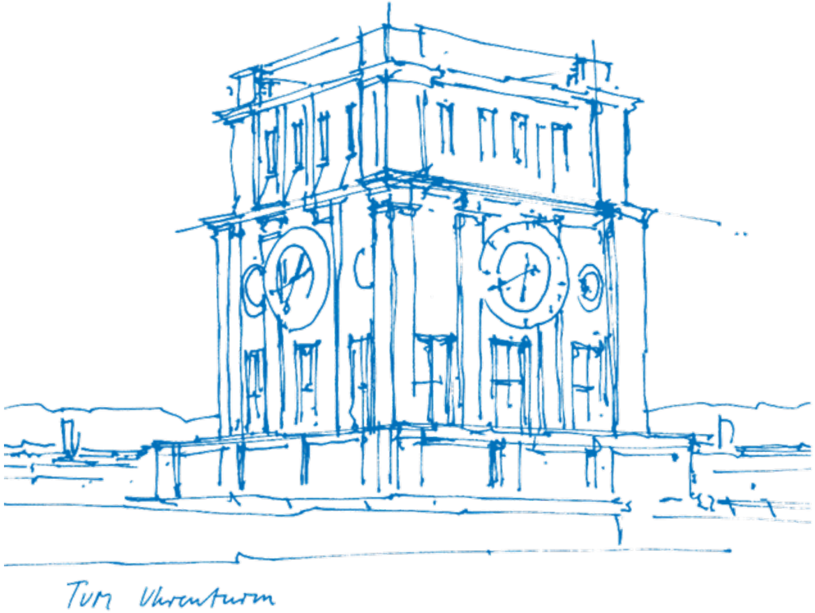
Parallel Programming Tutorial - Introduction to Pthread API

Bengisu Elis, M.Sc.
Chair for Computer Architecture and Parallel Systems (CAPS)
Technichal University Munich

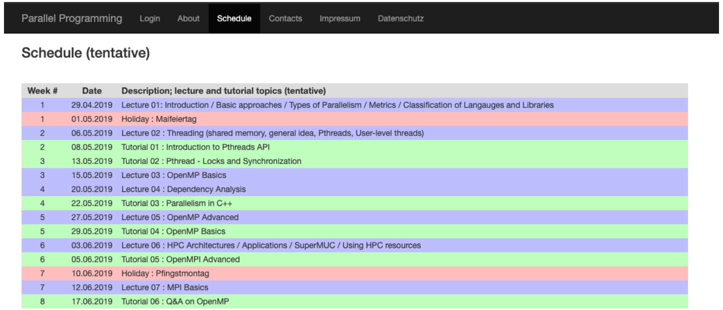
7. Mai 2019



Organization

Organization

- Course web-page
  - parprog.caps.in.tum.de
  - Register and login using your LRZId's (@mytum IDs).
  - Course schedule; lecture and tutorial
  - Exercises and assignment submission
  - Lecture and Tutorial slides are on Moodle !
- 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: bengisu.elis@tum.de
  - Room: MI, 01.04.053



Schedule (tentative)

Week #	Date	Description; lecture and tutorial topics (tentative)
1	29.04.2019	Lecture 01: Introduction / Basic approaches / Types of Parallelism / Metrics / Classification of Langauges and Libraries
1	01.05.2019	Holiday : Maifeiertag
2	06.05.2019	Lecture 02 : Threading (shared memory, general idea, Pthreads, User-level threads)
2	08.05.2019	Tutorial 01 : Introduction to Pthreads API
3	13.05.2019	Tutorial 02 : Pthread - Locks and Synchronization
3	15.05.2019	Lecture 03 : OpenMP Basics
4	20.05.2019	Lecture 04 : Dependency Analysis
4	22.05.2019	Tutorial 03 : Parallelism in C++
5	27.05.2019	Lecture 05 : OpenMP Advanced
5	29.05.2019	Tutorial 04 : OpenMP Basics
6	03.06.2019	Lecture 06 : HPC Architectures / Applications / SuperMUC / Using HPC resources
6	05.06.2019	Tutorial 05 : OpenMPI Advanced
7	10.06.2019	Holiday : Pfingstmontag
7	12.06.2019	Lecture 07 : MPI Basics
8	17.06.2019	Tutorial 06 : Q&A on OpenMP

Assignments

You have 1 week to complete each assignment

- We will work on 10 assignments on parallel programming techniques
- Submission of 80% of the assignments brings you 0.3 bonus
- Submission server:http://parprog.caps.in.tum.de
  - Walk through of the submission work-flow at the end of today's 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:

- Hasan Ashraf
  hasan.ashraf@tum.de

- Philipp Czerner
  philipp.czerner@tum.de

If you have questions regarding assignments and solutions, write an email to our tutors.

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++ (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 Command Line
- Resources
  - Book: The Linux Command Line
  - Basic video introduction: The Shell
- Knowing GCC
  - An Introduction to GCC, by Brian Gough

Posix Thread Programming

Posix Thread Programming

Definition: (Software) Thread

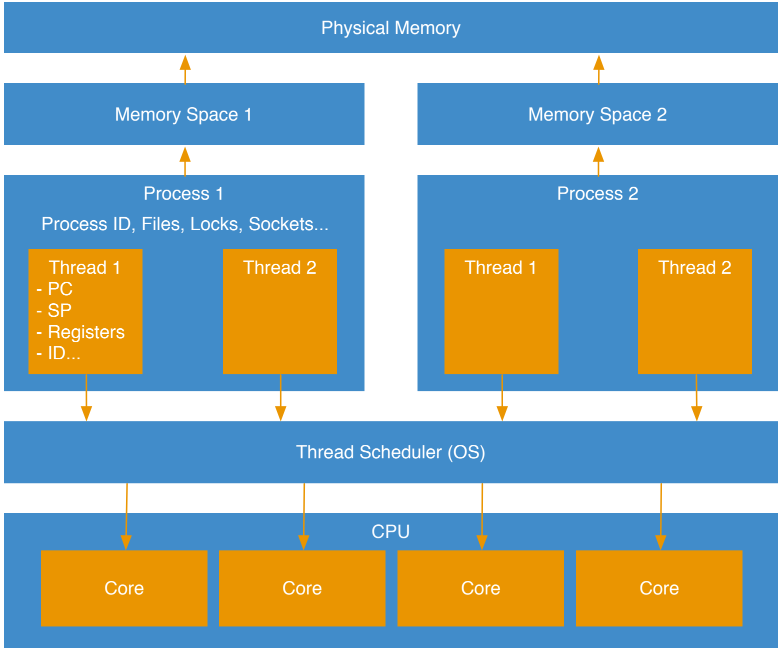
A thread is an independent stream of instructions that can be scheduled to run as such by the operating system. (Own PC and SP)

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



Pthread Syntax / Semantics

Create Pthreads

```c
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine) (void *),
                   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

```
int pthread_join(pthread_t thread,
                 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

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

// function to be executed by the thread
void* kernel (void* args){
  printf("hello from the thread!\n");
  return NULL;
}

int main(int argc, char *argv[])
{
  pthread_t thread;                                 // allocate a thread
  pthread_create(&thread, NULL, kernel , NULL);     // create the thread and start executing kernel in parallel to main thread
  printf("hello from main\n");
  pthread_join(thread,NULL);                        //wait for the thread to finish executing kernel

  return 0;
}
```

Compile & Output

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

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

Example 2; creating multiple threads

```c
int main(int argc, char *argv[])
{
  //allocate the threads
  int num_threads=4;
  pthread_t *threads = (pthread_t*) malloc (num_threads *sizeof(pthread_t));

  //create threads, start executing kernel in parallel
  for (int i = 0; i < num_threads; ++i) {
    pthread_create(&threads[i], NULL, kernel, NULL);
  }

  //wait for all the threads to finish executing kernel
  for (int i = 0; i < num_threads; ++i) {
    pthread_join(threads[i], NULL);
  }

  free(threads);
  return 0;
}
```

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

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

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

```c
int main(int argc, char *argv[])
{
  //allocate the threads
  int num_threads=4;
  pthread_t *threads = (pthread_t*) malloc (num_threads*sizeof(pthread_t));
  int* id = (int*) malloc (num_threads*sizeof(int));

  //create threads, start executing kernel in parallel
  for (int i = 0; i < num_threads; ++i) {
    id[i]=i; //set the id for the threads
    pthread_create(&threads[i], NULL, kernel, id+i); //pass the id as argument to the threads
  }

  //wait for all the threads to finish executing kernel
  for (int i = 0; i < num_threads; ++i) {
    pthread_join(threads[i], NULL);
  }

  free(threads); free(id);
  return 0;
}
```

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

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

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

```c
struct pthread_args
{
  long thread_id ;
  long num_threads ;
};

void* kernel (void* args){
  struct pthread_args *arg = (struct pthread_args*) args;
  printf("Hello from the thread, number of threads: %ld, myid: %ld, PID: %d, TID:%d!\n", \
  arg->num_threads, arg->thread_id, getpid(), (int) gettid());
  return NULL;
}
```

Example 5, passing multiple arguments (cont.)

```
int main(int argc, char *argv[])
{
  int num_threads=4;
  pthread_t *threads = (pthread_t*) malloc (num_threads*sizeof(pthread_t));
  struct pthread_args* args = (struct pthread_args*) malloc (num_threads*sizeof (struct pthread_args));

  for (int i = 0; i < num_threads; ++i) {
    //set the id and num threads in args for the threads
    args[i].thread_id=i;
    args[i].num_threads=num_threads;
    //pass the args as argument to the threads
    pthread_create(&threads[i], NULL, kernel, args+i); // passing args[i] to threads[i]
  }

  for (int i = 0; i < num_threads; ++i) {
    pthread_join(threads[i], NULL);
  }

  free(threads); free(args);
  return 0;
}
```

Example 6, how to get data out of threads

```
struct pthread_args
{
  int in ;
  int out ;
};

void* kernel_double (void* args){
  struct pthread_args *arg = (struct pthread_args*) args;
  arg->out = 2*arg->in;
  return NULL;
}
```

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

```c
int main(int argc, char *argv[])
{
  int num_threads=4;
  pthread_t *threads = (pthread_t*) malloc (num_threads*sizeof(pthread_t));
  struct pthread_args* args = (struct pthread_args*) malloc (num_threads*sizeof (struct pthread_args));

  for (int i = 0; i < num_threads; ++i) {
    args[i].in=i; //set the input in args
    pthread_create(&threads[i], NULL, kernel_double, args+i);
  }

  for (int i = 0; i < num_threads; ++i) {
    pthread_join(threads[i], NULL);
  }

  for (int i = 0; i < num_threads; ++i) {
    printf("Double of %d is %d!\n", args[i].in, args[i].out);
  }

  free (threads); free (args); return 0;
}
```

Example 7, return data from threads

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

Example 7, return data from threads (cont.)

```c
int main(int argc, char *argv[])
{
  int num_threads=4;
  pthread_t *threads = (pthread_t*) malloc (num_threads*sizeof(pthread_t));
  int* in = (int*) malloc (num_threads*sizeof(int));

  for (int i = 0; i < num_threads; ++i) {
    in[i]=i; //set the input for the threads
    pthread_create(&threads[i], NULL, kernel_double, in+i);
  }

  for (int i = 0; i < num_threads; ++i) {
    int *out;
    pthread_join(threads[i], (void*)&out);
    printf("Double of %d is %d!\n", in[i], *out);
    free(out);
  }

  free (threads); free (in); return 0;
}
```

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

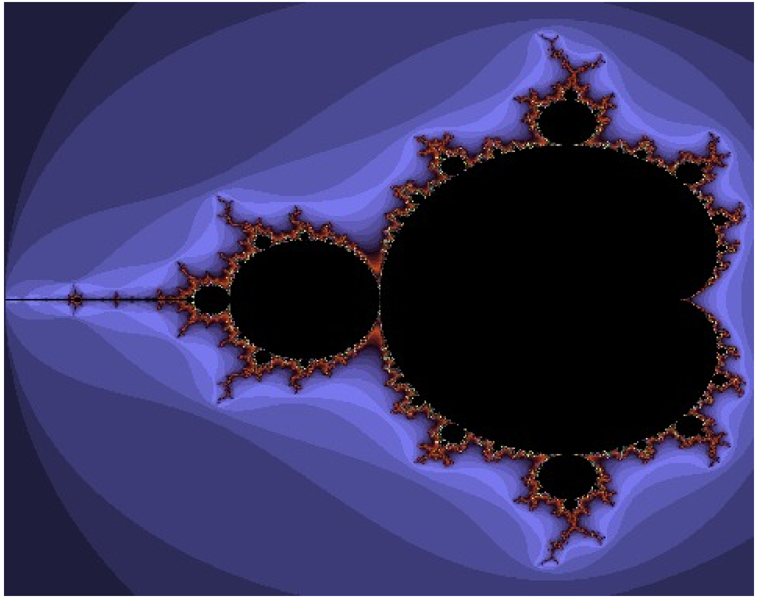
Assignment 1: “Mandelbrot set” in parallel

Assignment: Mandelbrot

Starting this week, you have one week time.

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

```
void mandelbrot_draw( ... some args ) {
  ...
  for (int i = 0; i < y_resolution; i++)
  {
    for (int j = 0; j < x_resolution; j++)
    {
      //embarrassingly parrallel calculation of pixels
      ...
    }
  }
}
```



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)

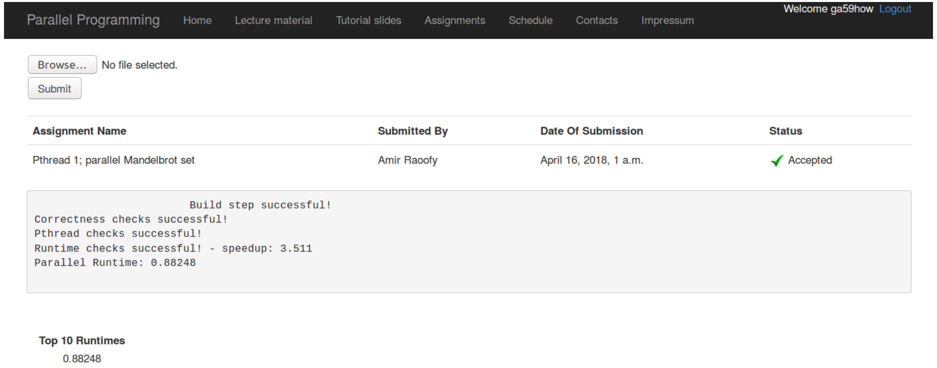
   ```
   student/mandelbrot_set_par -t 4
   ```

Are you a windows user?

- Install linux in VirtualBox
  - Don't forget to assign multiple cores to the virtual machine
- Use the Machines at Rechnerhalle
  - Use Putty
  - ssh server: lxhalle.informatik.tu-muenchen.de
  - You need to get access from info point in informatik if you already don't have an account
- Ask the tutors; they will be more than happy to help you.

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



No file selected.

Assignment Name	Submitted By	Date Of Submission	Status
Pthread 1; parallel Mandelbrot set	Amir Raoofy	April 16, 2018, 1 a.m.	✓ Accepted

```
                    Build step successful!
Correctness checks successful!
Pthread checks successful!
Runtime checks successful! - speedup: 3.511
Parallel Runtime: 0.88248
```

Top 10 Runtimes

0.88248