

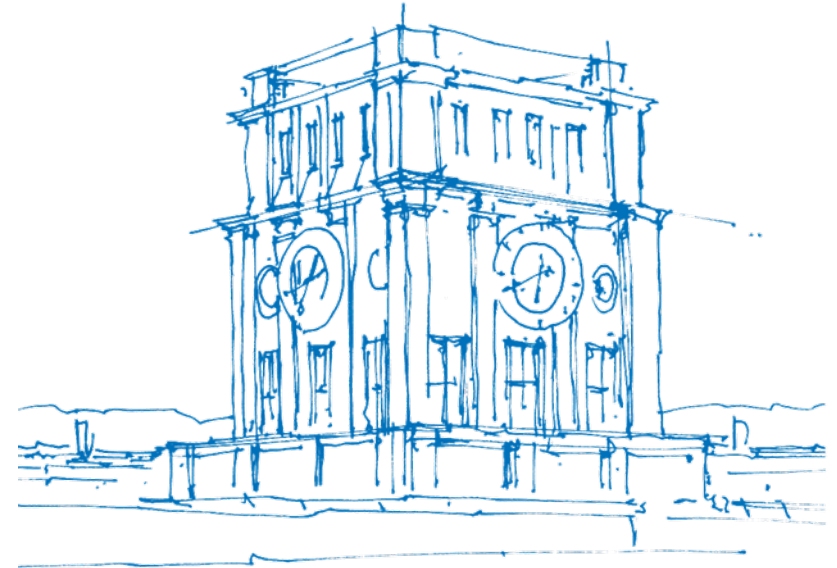
Parallel Programming Tutorial - More on Pthreads; synchronization

Amir Raoofy, M.Sc.

Chair for Computer Architecture and Parallel Systems (CAPS)

Technical University Munich

18. April 2018



TUM Uhrenturm

Organization re-visited

- We have (almost) one tutorial session per week.
- But: always check the schedule in the web-page: Next week we have two lectures and no tutorial.
- 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>
- You can only register until this Friday.
- QA Sessions are held in room 01.06.020 on Tuesdays 14:00 - 16:00 and 16:00 - 18:00.
- Tutors:
 - Canberk: canberk.demirsoy@tum.de
 - Rakesh: rakesh.singh@tum.de
 - Vishnu: vishnu.anilkumar-suma@tum.de
- My email address is: amir.raoofy@tum.de

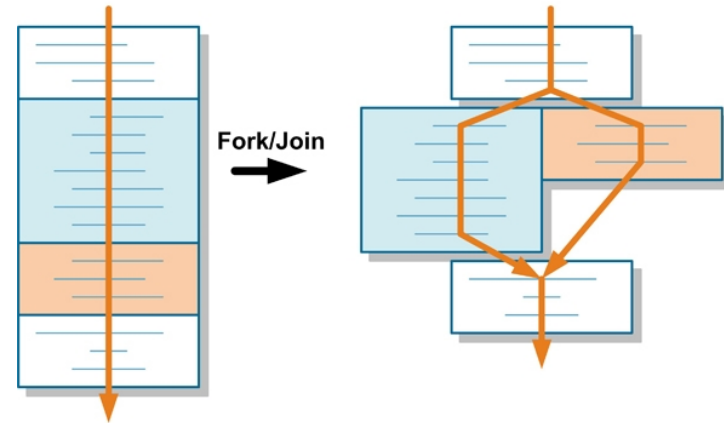
Are you a windows user?

- Install linux in VirtualBox
 - Dont 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 or go to the Q/A; they will be more than happy to help you.

Recap

Recap

- Pthread API
- Creating new threads with `pthread_create`
- Waiting for threads to finish with `pthread_join`
- Passing arguments to pthread kernels
- Returning results from pthread kernels

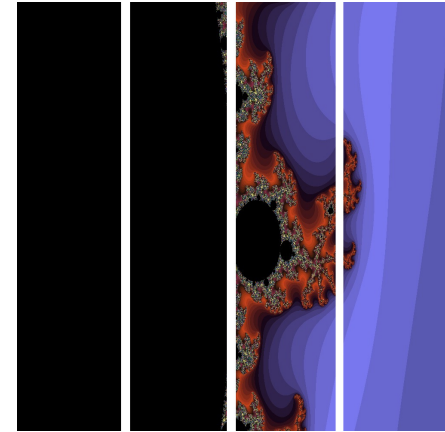
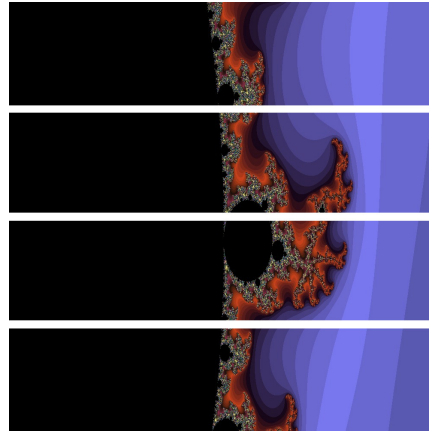
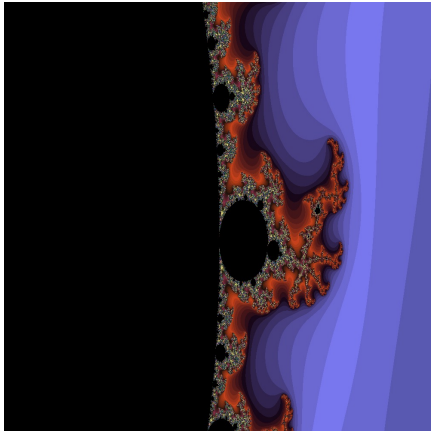


Tips for assignment 1

Hints for Assignment 1 / general Parallelization

- Consider static distribution
- Use a profiler to identify bottlenecks in your code
- Initialize Memory before it is used
- Consider false sharing to improve speedup
- Avoid data hazards during shared memory access
- Consider load balancing issues

Static (Block) Distribution



Profiling with perf

- perf can read performance event counters (HW counter)
- Install perf by `sudo apt-get install linux-tools-<kernel>`
- **Top:**
 - monitoring: `perf top`
- **Statistics:**
 - Collect and print statistics: `perf stat <cmd>`
 - Useful option: `-e <list of counters>` (see `perf help`)
- **Recording:**
 - To build call-graph with frame pointer, use gcc option `-fno-omit-frame-pointer`
 - To build call-graph with dwarf, use gcc option `-g`
 - Record with frame-pointer:
 - `perf record -g <cmd>`
 - Record with dwarf (for binaries without fp)
 - `perf record --call-graph dwarf <cmd>`
 - TUI: `perf report -G`

| | | | | |
|--------------|----------|----------------|------------------------|---------------------------------|
| Samples: 347 | of event | 'cycles:ppp' | Event count (approx.): | 30366896423 |
| Children | Self | Command | Shared Object | Symbol |
| + 90,22% | 30,61% | mandelbrot_set | mandelbrot_set_par | [.] compute_mandelbrot |
| + 94,19% | 0,00% | mandelbrot_set | [unknown] | [.] 0xffffffffffffffff |
| + 94,12% | 0,00% | mandelbrot_set | libc-2.23.so | [.] _clone |
| + 94,12% | 0,00% | mandelbrot_set | libpthread-2.23.so | [.] start_thread |
| + 41,61% | 39,80% | mandelbrot_set | libm-2.23.so | [.] _hypot_finite |
| + 40,15% | 1,37% | mandelbrot_set | libm-2.23.so | [.] _hypot |
| + 25,20% | 25,03% | mandelbrot_set | mandelbrot_set_par | [.] _muldc3 |
| + 2,41% | 0,00% | mandelbrot_set | [unknown] | [.] 0xbfe8f05d0eeb4c5a |
| + 2,11% | 2,11% | mandelbrot_set | [kernel.kallsyms] | [k] nmi |
| + 0,80% | 0,66% | mandelbrot_set | libm-2.23.so | [.] _cabs |
| + 0,41% | 0,00% | mandelbrot_set | [unknown] | [.] 0xbfb497603c920584 |
| + 0,12% | 0,00% | mandelbrot_set | [unknown] | [.] 0x3fa24f5fa5d5d4df |
| + 0,12% | 0,00% | mandelbrot_set | [unknown] | [.] 0xbfec9845830fd00b |
| + 0,10% | 0,00% | mandelbrot_set | [unknown] | [.] 0x3fb7eb6fca85cb7c |
| + 0,09% | 0,00% | mandelbrot_set | [unknown] | [.] 0xbfaaff46269df9a4f |
| + 0,09% | 0,09% | mandelbrot_set | mandelbrot_set_par | [.] cabs@plt |
| + 0,08% | 0,00% | mandelbrot_set | mandelbrot_set_par | [.] _start |
| + 0,08% | 0,00% | mandelbrot_set | libc-2.23.so | [.] __libc_start_main |
| + 0,08% | 0,00% | mandelbrot_set | mandelbrot_set_par | [.] main |
| + 0,08% | 0,00% | mandelbrot_set | [kernel.kallsyms] | [k] entry_SYSCALL_64_fastpath |
| + 0,07% | 0,00% | mandelbrot_set | libc-2.23.so | [.] _IO_file_xsputn@GLIBC_2.2.5 |
| + 0,07% | 0,00% | mandelbrot_set | [unknown] | [.] 0x3f94d30b1f36a04d |
| + 0,07% | 0,00% | mandelbrot_set | libc-2.23.so | [.] _IO_fwrite |
| + 0,07% | 0,00% | mandelbrot_set | libc-2.23.so | [.] _IO_file_write@GLIBC_2.2.5 |
| + 0,07% | 0,00% | mandelbrot_set | libc-2.23.so | [.] __GI__libc_write |
| + 0,07% | 0,00% | mandelbrot_set | [kernel.kallsyms] | [k] sys_write |
| + 0,07% | 0,00% | mandelbrot_set | [kernel.kallsyms] | [k] vfs_write |
| + 0,07% | 0,00% | mandelbrot_set | [kernel.kallsyms] | [k] __vfs_write |
| + 0,07% | 0,00% | mandelbrot_set | [kernel.kallsyms] | [k] new_sync_write |
| + 0,07% | 0,00% | mandelbrot_set | [kernel.kallsyms] | [k] ext4_file_write_iter |
| + 0,07% | 0,00% | mandelbrot_set | [kernel.kallsyms] | [k] __generic_file_write_iter |
| + 0,07% | 0,00% | mandelbrot_set | [kernel.kallsyms] | [k] generic_perform_write |

Initialization: malloc vs calloc

```
1 void *malloc(size_t size);
```

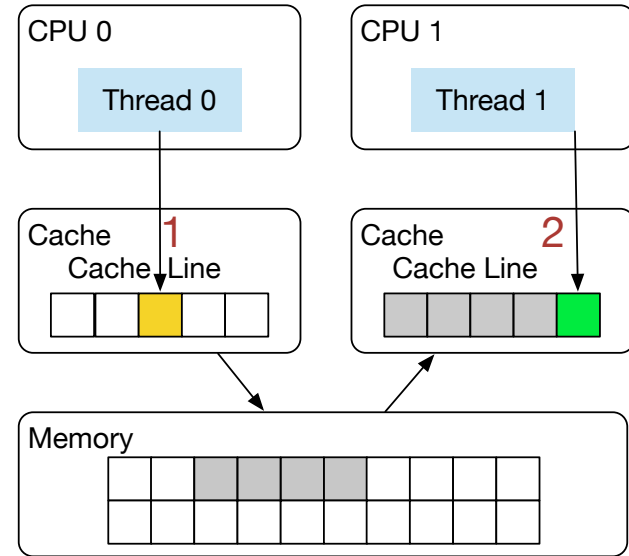
The `malloc()` function shall allocate unused space for an object whose size in bytes is specific by `size` and whose **value is unspecified**.

```
1 void *calloc(size_t nelem, size_t elsize);
```

The `calloc()` function shall allocate unused space for an array of `nelem` elements each of whose size in bytes is specific by `elsize`. **The space shall be initialized to all bits 0.**

False Sharing

- False sharing is a pattern that degrades performance
- It may appear on systems with distributed, coherent caches
- Multiple threads attempt to periodically access data that:
 - will never be altered by other threads
 - shares a cache line with data that is altered by other threads
- The caching protocol forces the first thread to reload the whole cache line
- Current compilers can detect false-sharing (use `-O2` in gcc)



False Sharing: Example

```
1  #include <sys/times.h>
2  #include <time.h>
3  #include <stdio.h>
4  #include <pthread.h>
5
6  #define NUM 100000000
7
8  int data [100];
9
10 void *kernel(void *args) {
11     int index = *(int *) args;
12     int i;
13     for (i = 0; i < NUM; i++)
14         data[index]++;
15 }
```

- kernel() increments an entry at location index in data by one.
- once the threads are started, they will access to neighboring elements in data
- consider the accesses to data elements regarding cache size.
- if both threads access the elements on the same cache-line we have false sharing

False Sharing: Example (Cont.)

```
1  int main(int argc, char *argv[]) {
2
3      long long int t1_start,t1_end,t2_start,t2_end;
4      float time1, time2;
5      pthread_t thread_1, thread_2;
6
7      int elem = 0, bad_elem = 1, good_elem = 32;
8
9      t1_start = clock();
10     pthread_create(&thread_1, NULL, kernel, &elem);
11     pthread_create(&thread_2, NULL, kernel, &bad_elem);
12     pthread_join(thread_1, NULL);
13     pthread_join(thread_2, NULL);
14     t1_end = clock();
15
16     t2_start = clock();
17     pthread_create(&thread_1, NULL, kernel, &elem);
18     pthread_create(&thread_2, NULL, kernel, &good_elem);
19     pthread_join(thread_1, NULL);
20     pthread_join(thread_2, NULL);
21     t2_end = clock();
22
23     time1 = (t1_end-t1_start)*1.0/CLOCKS_PER_SEC;
24     time2 = (t2_end-t2_start)*1.0/CLOCKS_PER_SEC;
25
26     printf("with false sharing:    %f sec\n", time1);
27     printf("without false sharing: %f sec\n", time2);
28
29     return 0;
30 }
```

`./false_sharing`

with false sharing: 1.497847 sec
without false sharing: 0.402427 sec

False Sharing: Yet another Example

- Problem:
 - `sizeof(thread_arg)` is less than size of cache-line
 - `malloc` allocates the structure instances subsequently
 - if `malloc` is used for allocation of the `thread_arg` then we might have false sharing
- Solutions:
 - Avoid subsequent memory blocks (each thread allocates own struct)
 - Add dummy attribute to structure that has at least size of cache line

```
1 typedef struct {
2     int num;
3     int i;
4 } thread_arg;
```



```
1 typedef struct {
2     int num;
3     int i;
4     char dummy[64];
5 } thread_arg;
```

Data Hazards

Data hazards occur when threads are accessing shared data. Ignoring potential data hazards can result in a race condition. There are three situations in which a data hazard can occur.

- read after write (RAW), a "true dependency"
- write after read (WAR), an "anti-dependency"
- write after write (WAW), an "output dependency"

Data Hazards: Incrementing i

```

1  #include <stdio.h>
2  #include <pthread.h>
3
4  #define NUM 10000000
5
6  // increment (*ptr) NUM times
7  void* increment(void *ptr) {
8
9      int *i = (int*)ptr;
10
11     for(int j=0; j < NUM; j++)
12         (*i)++;
13
14     return NULL;
15 }
16
17 int main(int argc, char** argv) {
18
19     int i = 0;
20     pthread_t thr;
21     pthread_create(&thr, NULL, &increment, &i);
22
23     for(int j=0; j < NUM; j++)
24         i++;
25
26     pthread_join(thr, NULL);
27     printf("Value of i = %d\n", i);
28
29     return 0;
30 }

```

```

$ ./increment_integer
Value of i = 185363257
$ time ./increment_integer
Value of i = 181870167

```

```

real    0m0.517s
user    0m0.491s
sys     0m0.008s

```


Data Hazards: Incrementing i (Cont.)

Problem

- $i++$ is no atomic operation
 1. `load R1, (x);` - Fetch i into a register
 2. `add R1, R1, #1;` - Increment the register
 3. `store (x), R1;` - Write back to i
- Different threads may interrupt at any point

Possible Solutions

- Avoid data hazards (e.g., by reduction)
- Use synchronization

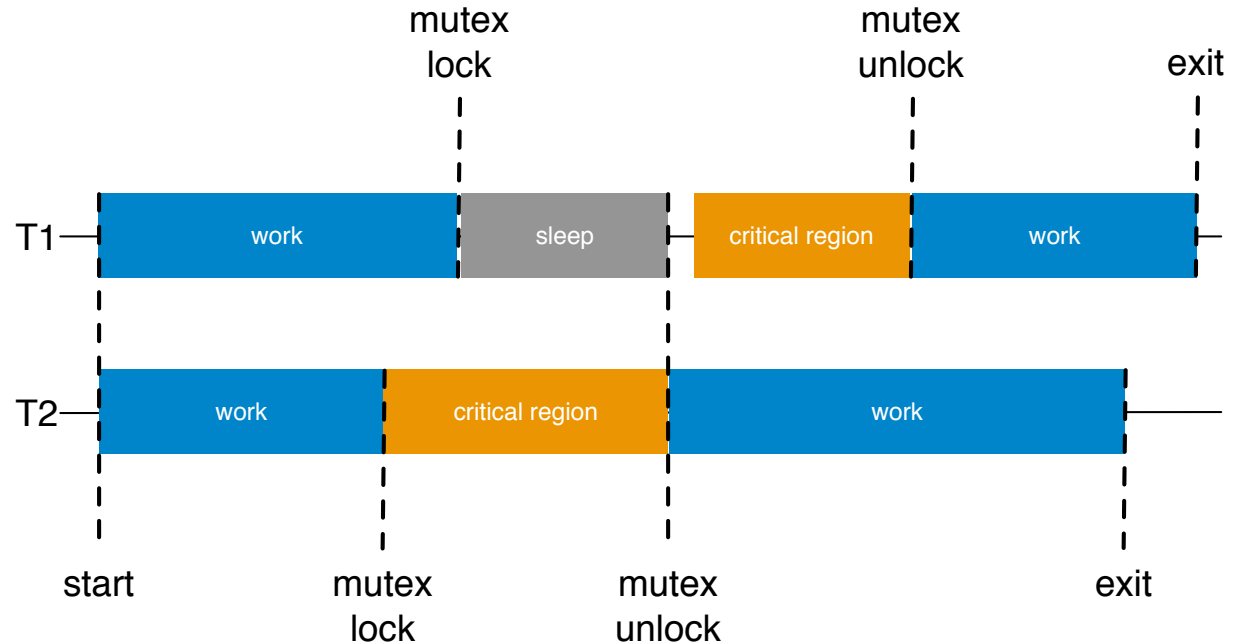
Synchronization with Pthread

Synchronization

- Synchronization needed for accesses to shared data and resources
- Drawback: Serializes applications
- The mostly used operations for synchronization are:
 - Mutual exclusion of critical regions
 - Conditional synchronization
- Pthread provides following mechanisms:
 - Mutexes
 - Condition Variables (not covered)
 - Barriers (not covered)
 - Semaphores (not covered)

Mutexes (mutual exclusion lock)

- The simplest and most primitive synchronization variable
- Implemented by using atomic (hardware) operations
- Provides an absolute owner for a code (critical) section
- Threads can lock and unlock mutexes



Mutexes: Creation and Destroying

```
1 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;  
2 int pthread_mutex_init( pthread_mutex_t *mutex,  
3                        pthread_mutexattr_t *attr );  
4 int pthread_mutex_destroy( pthread_mutex_t *mutex );
```

- `pthread_mutex_t *mutex`
 - Pointer to a mutex.
- `pthread_mutexattr_t *attr`
 - Optional pointer to `pthread_mutexattr_t` to define behavior, if NULL defaults are used.
 - Options: [Cross-Process, Priority-Inheriting]
- Use static initialization for static mutexes with default attributes (you do not have to destroy a static mutex)
- Use dynamic initialization for malloc and non-standard attributes (destroying of the mutex necessary)

Mutexes: Locking and Unlocking

```
1 int pthread_mutex_lock( pthread_mutex_t *mutex );
2 int pthread_mutex_trylock( pthread_mutex_t *mutex );
3 int pthread_mutex_unlock( pthread_mutex_t *mutex );
```

- `pthread_mutex_t *mutex`
 - Pointer to a mutex.
- A thread can lock an unlocked mutex → **it owns the mutex**
- If a thread wants to lock a locked mutex, the calling thread blocks until the mutex is available.
- `pthread_mutex_trylock` locks the mutex if it is unlocked. Otherwise it returns `EBUSY`
- A thread may unlock a mutex that it owns and blocked threads will be awakened.
- Threads cannot unlock mutexes that they do not own or that are already unlocked (error)
- Recursive locking behavior depends on the type of mutex:
 - `PTHREAD_MUTEX_NORMAL` (deadlock), `PTHREAD_MUTEX_ERRORCHECK` (error code), `PTHREAD_MUTEX_RECURSIVE` (lock count),
default: undefined behavior

Mutexes: Example

```
1  #include <stdio.h>
2  #include <pthread.h>
3
4  #define NUM 10000000
5
6  pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
7
8  void * increment(void *i_void_ptr)
9  {
10     int *i = (int *) i_void_ptr;
11
12     for(int j=0; j < NUM; j++)
13     {
14         pthread_mutex_lock(&mutex);
15         (*i)++;
16         pthread_mutex_unlock(&mutex);
17     }
18
19     return NULL;
20 }
```

Mutexes: Example (Cont.)

```
1  int main(int argc, char** argv)
2  {
3      int i = 0;
4      pthread_t thr;
5      pthread_create(&thr, NULL, &increment, &i);
6
7      for(int j=0; j < NUM; j++) {
8          pthread_mutex_lock(&mutex);
9          i++;
10         pthread_mutex_unlock(&mutex);
11     }
12
13     pthread_join(thr, NULL);
14     printf("Value of i = %d\n", i);
15
16     return 0;
17 }
```

```
time ./incrementi_mutex
Value of i = 200000000
```

```
real  0m4.659s
user  0m4.366s
sys   0m0.033s
```


Reduction: Example

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4
5  #define NUM 10000000
6
7  void * increment(void *ptr)
8  {
9      int *i = malloc( sizeof(int) );
10
11     for(int j=0; j < NUM; j++)
12         (*i)++;
13
14     return i;
15 }

```

avoid



- reduce the single results into a (fresh allocated) variable
- return the variable

Reduction: Example (Cont.)

```

1  int main(int argc, char** argv)
2  {
3      int i_1 = 0;
4      void *i_2;
5      pthread_t thr;
6      pthread_create(&thr, NULL, &increment, NULL);
7
8      for(int j=0; j < NUM; j++)
9          i_1++;
10
11     pthread_join(thr, &i_2);
12     i_1 += *((int*)i_2);
13
14     // important since the thread allocated memory
15     free(i_2);
16     printf("Value of i = %d\n", i_1);
17     return 0;
18 }
```

```

time ./incrementi_reduction
Value of i = 200000000
```

```

real    0m0.531s
user    0m0.502s
sys     0m0.004s
```

Atomic Variables (C++11): Example

```
1 #include <stdio.h>
2 #include <atomic.h>           // important
3 #include <pthread.h>
4
5 #define NUM 10000000
6
7 void * increment(void *ptr)
8 {
9     std::atomic<int> *i = (std::atomic<int>*)ptr;
10
11     for(int j=0; j < NUM; j++)
12         (*i)++;
13
14     return NULL;
15 }
```

- `std::atomic<int>` is a specialization (`int`) of the atomic template in C++11.
- Might use atomic operations or locking depending on compiler/hardware.

Atomic Variables(C++11): Example (Cont.)

```

1  int main(int argc, char** argv) {
2      std::atomic<int> i;
3      i=0;
4      pthread_t thr;
5      pthread_create(&thr, NULL, &increment, &i);
6
7      for(int j=0; j < NUM; j++)
8          i++;
9
10     pthread_join(thr, NULL);
11     printf("Value of i = %d\n", i.load());
12
13     return 0;
14 }
```

```
time ./atomic_variables
Value of i = 20000000
```

```
real    0m0.377s
user    0m0.699s
sys     0m0.000s
```



Extended Topics: Synchronization

- **Spinlocks**



```
int spin_lock( spinlock_t* )  
int pthread_spin_lock( pthread_spin_lock_t* )  
– Spinlocks do not block, but "spin"  
– Benefit: no context switches, good for fine-grained locking  
– Drawback: may cause deadlock on a single-core processor  
– First/Second version allows synchronization on processes/threads
```

- **Recursive Mutexes**



```
mutexattr = PTHREAD_MUTEX_RECURSIVE  
– Thread can relock a mutex it owns  
– Can be useful for making old interfaces thread-safe
```

- **Read/write locks**

```
int rwl_init( rwlock_t *rwlock )  
int rwl_readlock( rwlock_t *rwlock )  
int rwl_writelock( rwlock_t *rwlock )  
...
```

Extended Topics: Synchronization (Cont.)

- **Semaphores**

- API: `sem_init(...)`, `sem_post(...)` and `sem_wait(...)`
- `sem_init` initializes semaphore with value `x`
- `sem_post` post a wakeup to a semaphore. If there are waiting threads, one is awakened. Otherwise, the semaphore value is incremented by one.
- `sem_wait` wait on a semaphore. If the semaphore value is greater than 0, decrease the value by one. Otherwise, the thread is blocked.

- **Barriers**

Barrier is a point in the program where we want all the threads to reach it before continuing.

Useful for synchronization of threads

Also Useful for time measurement of the threaded regions

- API: `pthread_barrier_init(...)`, `pthread_barrier_destroy(...)` and `barrier_wait(...)`



Extended Topics: Barrier

```

1  #include <unistd.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  #include <stdio.h>
5
6  pthread_barrier_t barrier;
7
8  /* this function is executed by each thread */
9  void* kernel(void* arg) {
10
11     int id = * (int*) arg;
12
13     printf("Thread %d starting!\n", id);
14     // each thread waits for different amount of time
15     sleep(id);
16     printf("Thread %d is done!\n", id);
17
18     //barrier
19     pthread_barrier_wait(&barrier);
20
21     printf("Thread %d past the barrier!\n", id);
22     return NULL;
23 }

```

```

1  int main (int argc, char** argv) {
2
3     int num_threads=4;
4
5     pthread_t threads[num_threads];
6     int ids[num_threads];
7
8     pthread_barrier_init(&barrier, NULL, num_threads);
9
10    /* spawn the threads */
11    for (int i = 0; i < num_threads; i++) {
12        ids[i] = i;
13        pthread_create(threads+i, NULL, kernel, ids+i);
14    }
15
16    /* join all threads */
17    for (int i = 0; i < num_threads; i++) {
18        pthread_join(threads[i], NULL);
19    }
20
21    pthread_barrier_destroy(barrier);
22
23    return 0;
24 }
25

```

Extended Topics: Condition Variable

```

1  int num_threads=4;
2  int counter = 0;
3  pthread_cond_t condition;
4  pthread_mutex_t condition_mutex;
5
6  void* kernel(void* arg) {
7      int id = * (int*) arg;
8
9      printf("Thread %d starting!\n", id);
10     sleep(id);
11     printf("Thread %d is done!\n", id);
12
13     //implementation of barrier using condition variables
14     pthread_mutex_lock(&condition_mutex);
15     counter++;
16
17     if (counter == num_threads) {
18         counter = 0;
19         pthread_cond_broadcast(&condition);
20     } else {
21         pthread_cond_wait(&condition, &condition_mutex);
22     }
23     pthread_mutex_unlock(&condition_mutex);
24
25     printf("Thread %d past the barrier!\n", id);
26     return NULL;
27 }

```

```

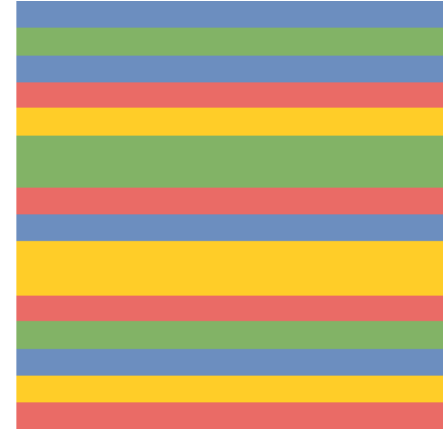
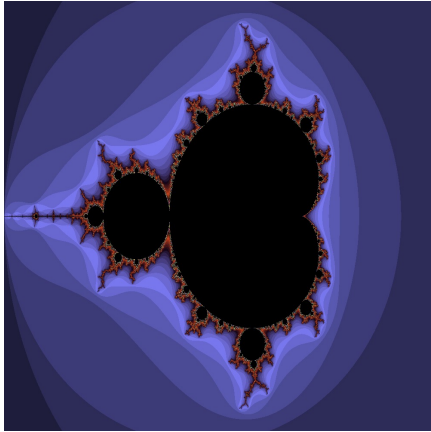
1  #include <unistd.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  #include <stdio.h>
5
6  int main (int argc, char** argv) {
7
8      pthread_t threads[num_threads];
9      int ids[num_threads];
10
11     /* spawn the threads */
12     for (int i = 0; i < num_threads; i++) {
13         ids[i] = i;
14         pthread_create(threads+i, NULL, kernel, ids+i);
15     }
16
17     /* join all threads */
18     for (int i = 0; i < num_threads; i++) {
19         pthread_join(threads[i], NULL);
20     }
21
22     return 0;
23 }

```


Assignment 2 - Mandelbrot set (dynamic)

Assignment 2 - Mandelbrot set (dynamic) - motivation

- Towards load balancing, cyclic and dynamic distribution of the loop



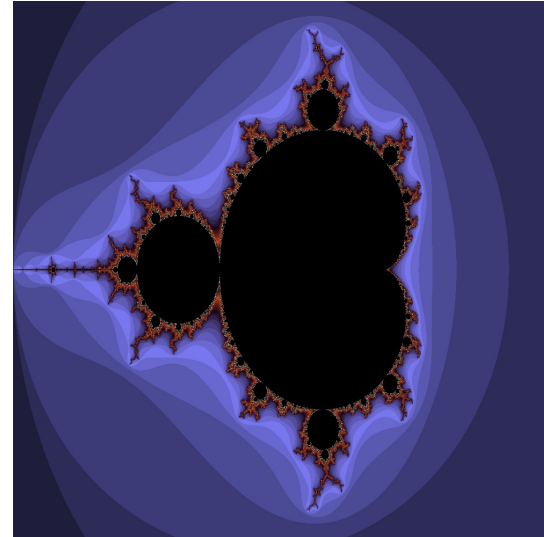
Assignment 2 - Mandelbrot set (dynamic)

- Use POSIX threads to parallelize `draw_mandelbrot(...)`
- The program should follow the producer/consumer pattern:
 - There is one producer thread (the main thread) that prepares chunks of images as work for threads.
 - There are `num_threads` worker threads that use the chunks as input and calculate values of pixels asynchronously.
 - As soon as chunks are available, a free worker grabs it and begins calculate the pixel values.
 - The results are used by the producer thread to progress.
- Consider:
 - The number of created threads are checked.
 - You may have to use shared variables, think about synchronization.
 - The speedup with 32 (logical) cores must be at least 14 (we might update the requirements).

Assignment 2 - Mandelbrot set (dynamic)

Starting this week, you have two weeks time.

```
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 2 - Mandelbrot set (dynamic) - 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_set_seq.c
 - Sequential version of draw_mandelbrot()
- student/mandelbrot_set_par.c
 - Implement the parallel version in this file
- unit_test.c
 - The unit tests that execute both the serial and parallel version to compare results.

Assignment 2 - Mandelbrot set (dynamic) Extract, Build and Run

1. Extract all files to the current directory

```
tar -xvf assignment2.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
```