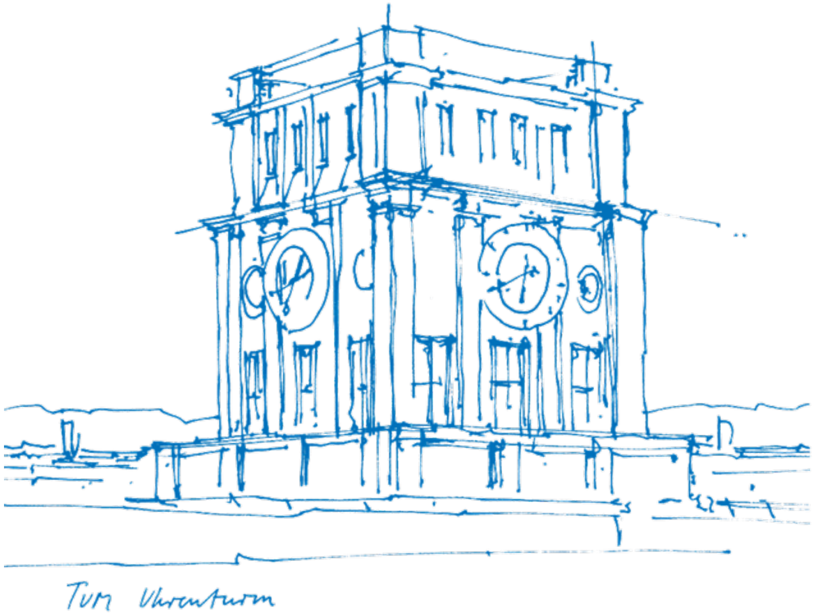
Parallel Programming Tutorial - More on Pthreads; synchronization

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

18. April 2018



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(at)tum.de
  - Rakesh: rakesh.singh(at)tum.de
  - Vishnu: vishnu.anilkumar-suma(at)tum.de
- My email address is: amir.raoofy(at)tum.de

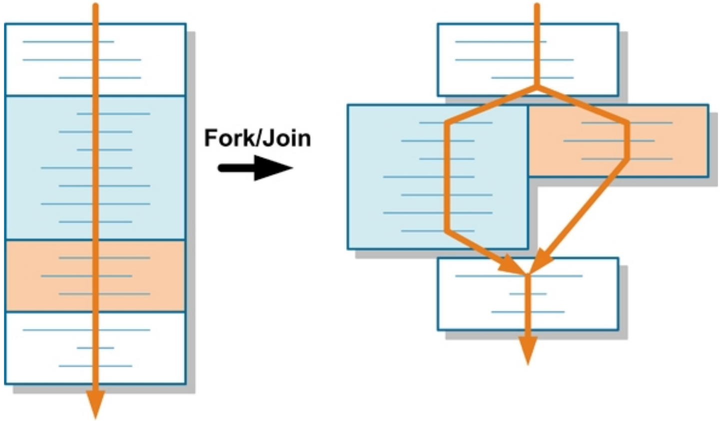
Are you a windows user?

- 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

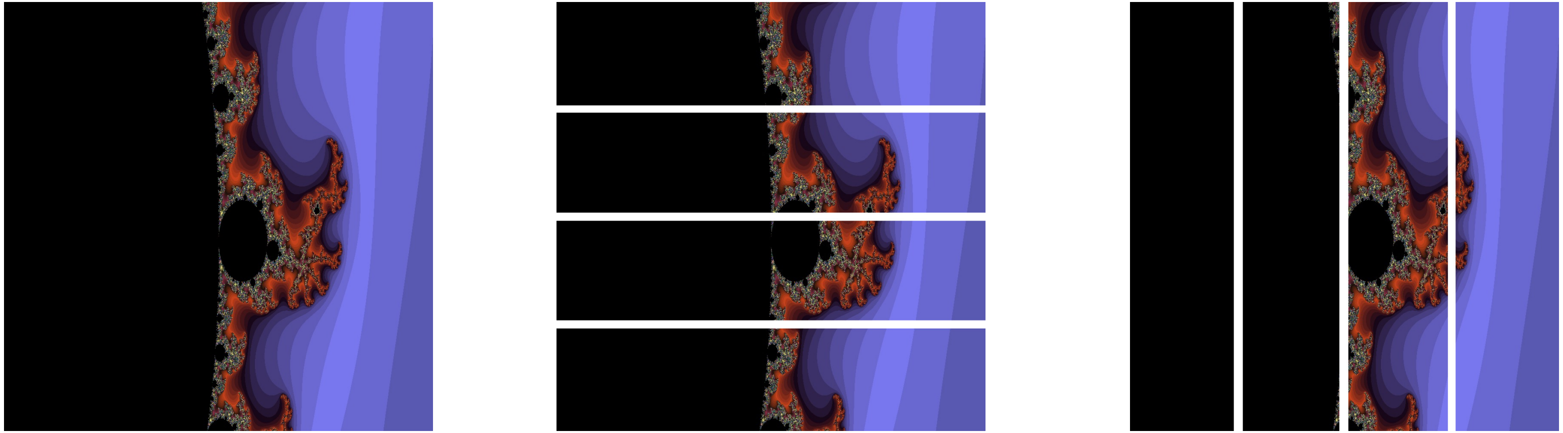


Tips for assignment 1

Hints for Assignment 1 / general Parallelization

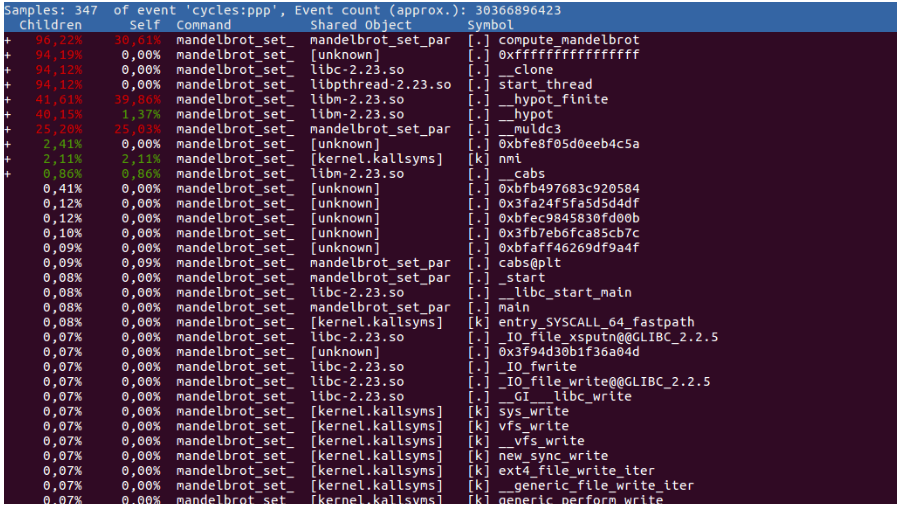
- Consider static distribution
- Use a profiler to identify bottlenecks in your code
- Initialize Memory before it is used
- Consider false sharing to improve speedup
- Avoid data hazards during shared memory access
- Consider load balancing issues

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
+   96,22%    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%    39,86%  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
+    0,86%     0,86%  mandelbrot_set_   libm-2.23.so          [.] __cabs
     0,41%     0,00%  mandelbrot_set_   [unknown]             [.] 0xbfb497683c920584
     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]             [.] 0xbfaff46269df9a4f
     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`

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

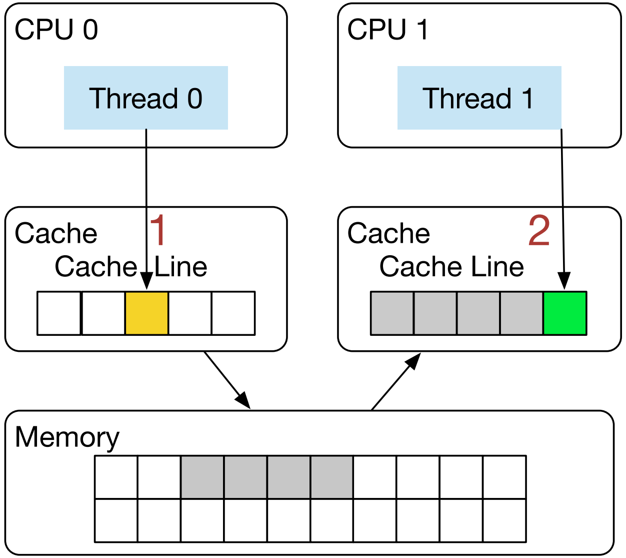
The `malloc()` function shall allocate unused space for an object whose size in bytes is specific by `size` and whose **value is unspecified**.

```
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:
  1. will never be altered by other threads
  2. 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

```c
#include <sys/times.h>
#include <time.h>
#include <stdio.h>
#include <pthread.h>

#define NUM 100000000

int data [100];

void *kernel(void *args) {
  int   index =  *(int *) args;
  int   i;
  for (i = 0; i < NUM; i++)
    data[index]++;
}
```

- `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.)

```c
int main(int argc, char *argv[]) {

  long long int t1_start,t1_end,t2_start,t2_end;
  float time1, time2;
  pthread_t thread_1, thread_2;

  int elem = 0, bad_elem = 1, good_elem = 32;

  t1_start = clock();
  pthread_create(&thread_1, NULL, kernel, &elem);
  pthread_create(&thread_2, NULL, kernel, &bad_elem);
  pthread_join(thread_1, NULL);
  pthread_join(thread_2, NULL);
  t1_end = clock();

  t2_start = clock();
  pthread_create(&thread_1, NULL, kernel, &elem);
  pthread_create(&thread_2, NULL, kernel, &good_elem);
  pthread_join(thread_1, NULL);
  pthread_join(thread_2, NULL);
  t2_end = clock();

  time1 = (t1_end-t1_start)*1.0/CLOCKS_PER_SEC;
  time2 = (t2_end-t2_start)*1.0/CLOCKS_PER_SEC;

  printf("with false sharing:    %f sec\n", time1);
  printf("without false sharing: %f sec\n", time2);

  return 0;
}
```

```
./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

```c
typedef struct {
  int num;
  int i;
} thread_arg;
```

- Solutions:
  - Avoid subsequent memory blocks (each thread allocates own struct)
  - Add dummy attribute to structure that has at least size of cache line

```c
typedef struct {
  int num;
  int i;
  char dummy[64];
} 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

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

#define NUM 10000000

// increment (*ptr) NUM times
void* increment(void *ptr) {

  int *i = (int*)ptr;

  for(int j=0; j < NUM; j++)
    (*i)++;

  return NULL;
}

int main(int argc, char** argv) {

  int i = 0;
  pthread_t thr;
  pthread_create(&thr, NULL, &increment, &i);

  for(int j=0; j < NUM; j++)
    i++;

  pthread_join(thr, NULL);
  printf("Value of i = %d\n", i);

  return 0;
}
```

```
$ ./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

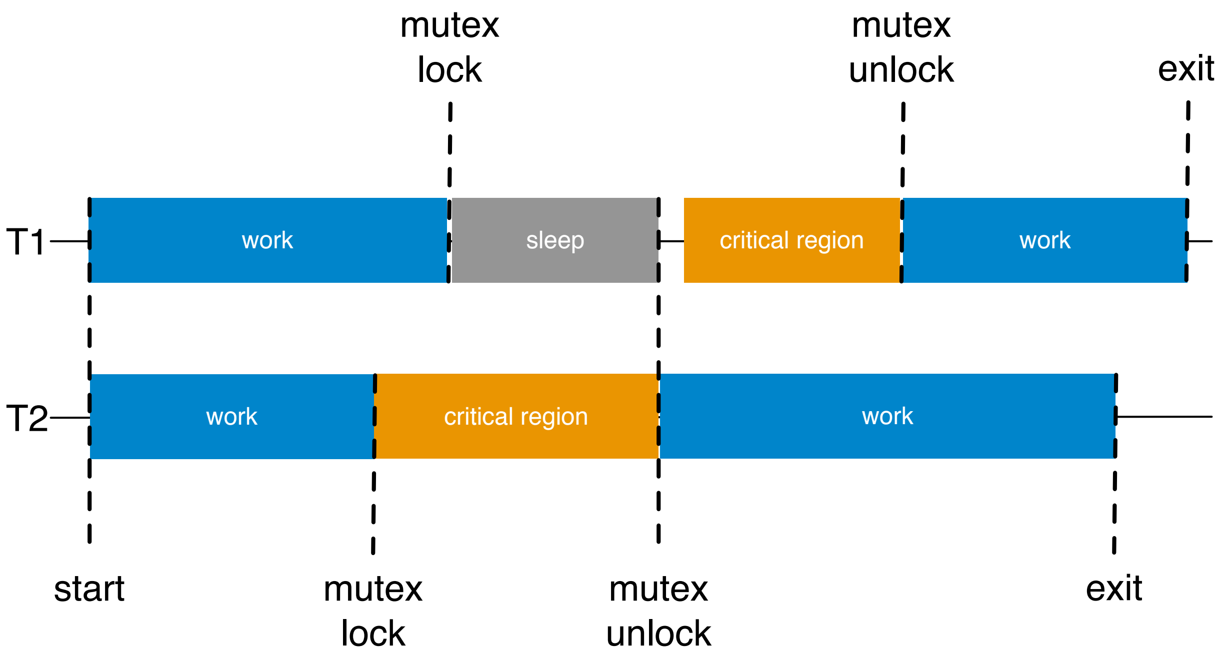
Synchronization with Pthread

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

```c
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_init( pthread_mutex_t *mutex,
                        pthread_mutexattr_t *attr );
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

```c
int pthread_mutex_lock( pthread_mutex_t *mutex );
int pthread_mutex_trylock( pthread_mutex_t *mutex );
int pthread_mutex_unlock( pthread_mtuex_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

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

#define NUM 10000000

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void * increment(void *i_void_ptr)
{
  int *i = (int *) i_void_ptr;

  for(int j=0; j < NUM; j++)
  {
    pthread_mutex_lock(&mutex);
    (*i)++;
    pthread_mutex_unlock(&mutex);
  }

    return NULL;
}
```

Mutexes: Example (Cont.)

```c
int main(int argc, char** argv)
{
  int i = 0;
  pthread_t thr;
  pthread_create(&thr, NULL, &increment, &i);

  for(int j=0; j < NUM; j++) {
    pthread_mutex_lock(&mutex);
    i++;
    pthread_mutex_unlock(&mutex);
  }

  pthread_join(thr, NULL);
  printf("Value of i = %d\n", i);

  return 0;
}
```

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

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

Reduction: Example

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

#define NUM 10000000

void * increment(void *ptr)
{
  int *i = malloc( sizeof(int) );

  for(int j=0; j < NUM; j++)
    (*i)++;

  return i;
}
```

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

Reduction: Example (Cont.)

```c
int main(int argc, char** argv)
{
  int i_1 = 0;
  void *i_2;
  pthread_t thr;
  pthread_create(&thr, NULL, &increment, NULL);

  for(int j=0; j < NUM; j++)
    i_1++;

  pthread_join(thr, &i_2);
  i_1 += *((int*)i_2);

  // important since the thread allocated memory
  free(i_2);
  printf("Value of i = %d\n", i_1);
  return 0;
}
```

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

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

Atomic Variables (C++11): Example

```
#include <stdio.h>
#include <atomic.h>        // important
#include <pthread.h>

#define NUM 10000000

void * increment(void *ptr)
{
  std::atomic<int> *i = (std::atomic<int>*)ptr;

  for(int j=0; j < NUM; j++)
    (*i)++;

  return NULL;
}
```

- `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.)

```
nt main(int argc, char** argv) {
  std::atomic<int> i;
  i=0;
  pthread_t thr;
  pthread_create(&thr, NULL, &increment, &i);

  for(int j=0; j < NUM; j++)
    i++;

  pthread_join(thr, NULL);
  printf("Value of i = %d\n", i.load());

  return 0;
}
```

```
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

```
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdio.h>

pthread_barrier_t barrier;

/* this function is executed by each thread */
void* kernel(void* arg) {

  int id = * (int*) arg;

  printf("Thread %d starting!\n", id);
  // each thread waits for different amount of time
  sleep(id);
  printf("Thread %d is done!\n", id);

  //barrier
  pthread_barrier_wait(&barrier);

  printf("Thread %d past the barrier!\n", id);
  return NULL;
}
```

```
int main (int argc, char** argv) {

  int num_threads=4;

  pthread_t threads[num_threads];
  int ids[num_threads];


  pthread_barrier_init(&barrier, NULL, num_threads);

  /* spawn the threads */
  for (int i = 0; i < num_threads; i++) {
    ids[i] = i;
    pthread_create(threads+i, NULL, kernel, ids+i);
  }

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

  pthread_barrier_destroy(barrier);

  return 0;
}
```

Extended Topics: Condition Variable

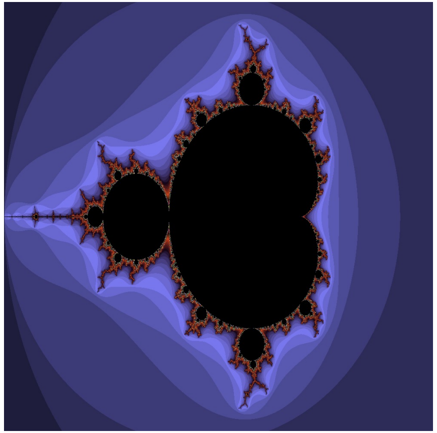
```c
int num_threads=4;
int counter = 0;
pthread_cond_t condition;
pthread_mutex_t condition_mutex;

void* kernel(void* arg) {
  int id = * (int*) arg;

  printf("Thread %d starting!\n", id);
  sleep(id);
  printf("Thread %d is done!\n", id);

  //implementation of barrier using condition variables
  pthread_mutex_lock(&condition_mutex);
  counter++;

  if (counter == num_threads) {
    counter = 0;
    pthread_cond_broadcast(&condition);
  } else {
    pthread_cond_wait(&condition, &condition_mutex);
  }
  pthread_mutex_unlock(&condition_mutex);

  printf("Thread %d past the barrier!\n", id);
  return NULL;
}
```

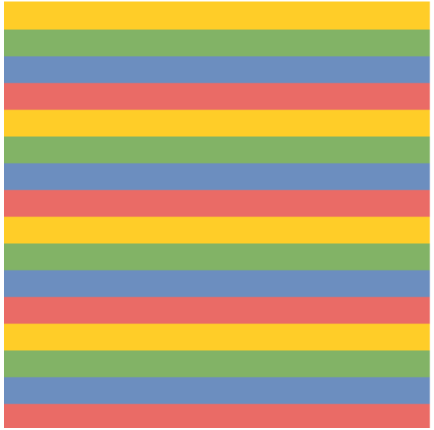
```c
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdio.h>

int main (int argc, char** argv) {

  pthread_t threads[num_threads];
  int ids[num_threads];

  /* spawn the threads */
  for (int i = 0; i < num_threads; i++) {
    ids[i] = i;
    pthread_create(threads+i, NULL, kernel, ids+i);
  }

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

  return 0;
}
```

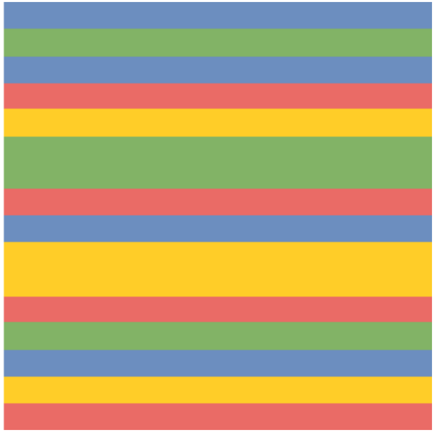
Assignment 2 - Mandelbrot set (dynamic)

Assignment 2 - Mandelbrot set (dynamic) - motivation

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







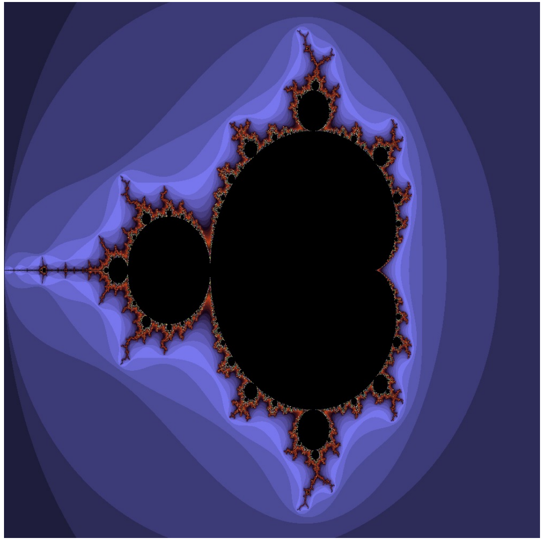
Assignment 2 - Mandelbrot set (dynamic)

- 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 on 32 (logical) cores must be at least 16.0 using 32 threads.

Assignment 2 - Mandelbrot set (dynamic)

Starting this week, you have two weeks time.

```
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 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`