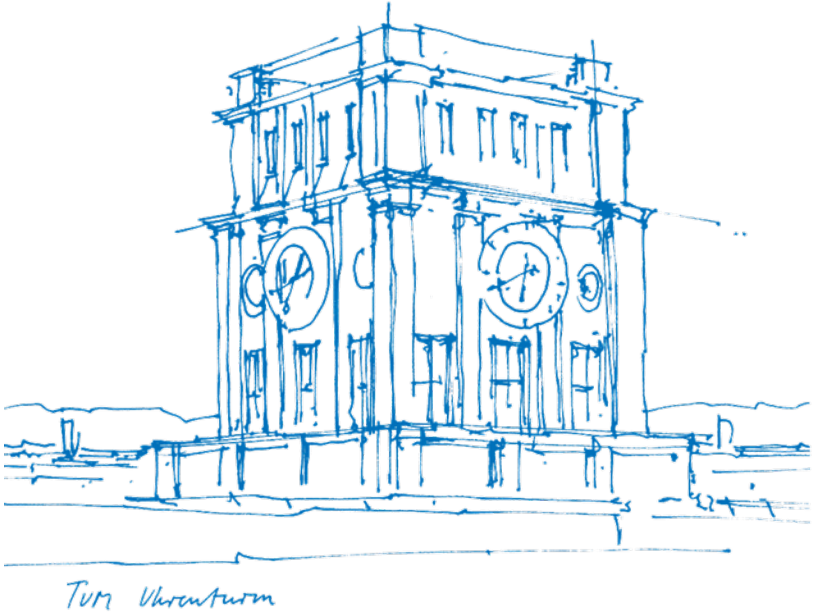
Parallel Programming Tutorial - Pthread 2

M.Sc. Andreas Wilhelm
Technichal University Munich

May 15, 2017



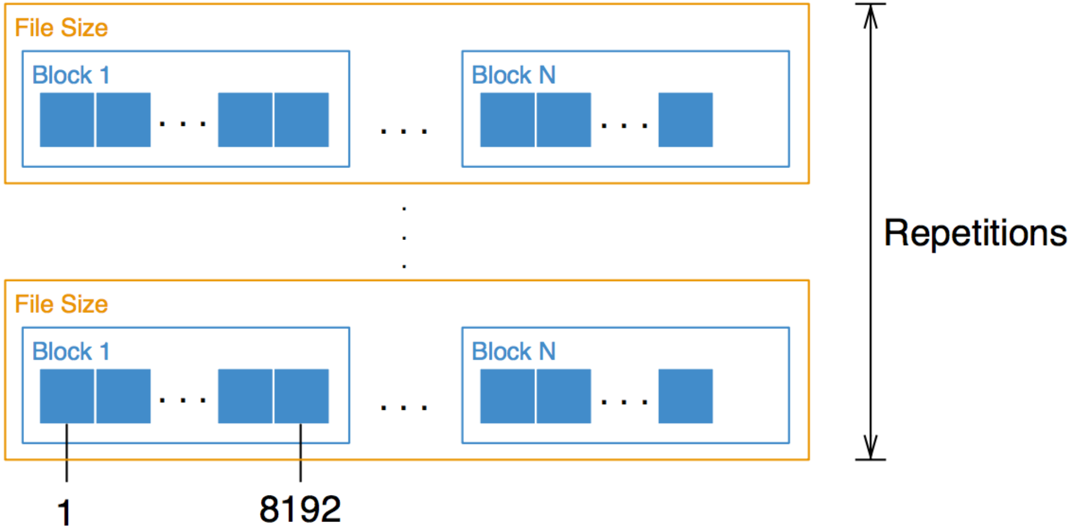
Assignment 1

Hints for Assignment 1 / general Parallelization

- Consider static distribution
- Use a profiler to identify best parallelization approach
- Initialize Memory before it is used
- Consider false sharing to improve speedup
- Avoid data hazards during shared memory access

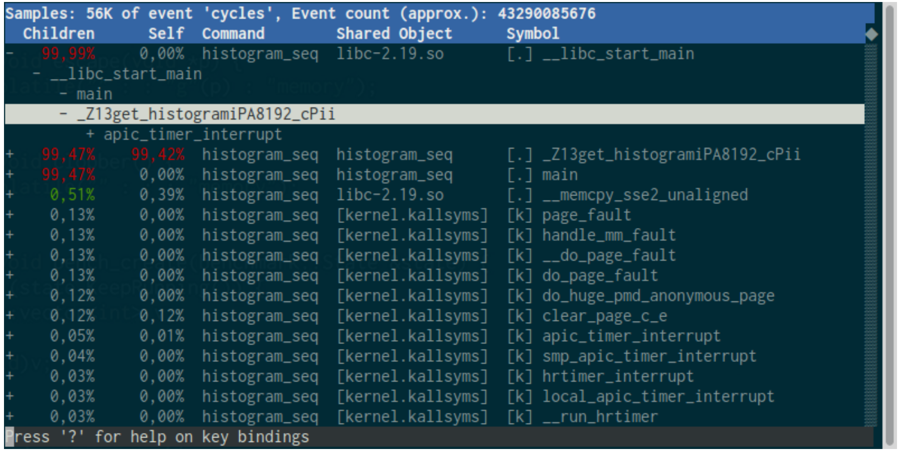
Static Distribution

- Distribute all! blocks across thre threads
- Consider that *nBlocks* % *num_threads* may be greater 0



Profiling with perf

- perf can read performance event counters (HW counter)
- Install perf by sudo apt-get install linux-tools-<kernel>
- **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



Initialization: `malloc` vs `calloc`

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

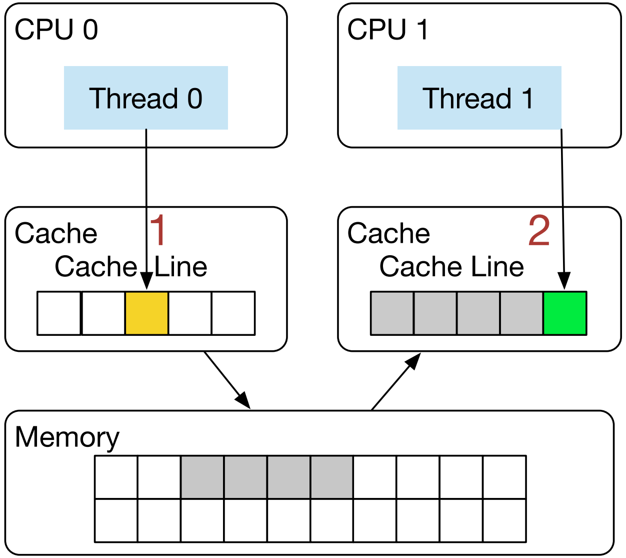
The `malloc()` function shall allocate unused space for an object whose size in bytes is specific by `size` and whose **value is unspecified**.

```c
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 (1/3)

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

#define NTHREADS 2
#define NUM 1000000000

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

void* increment( void* ptr ) {

  thread_arg* arg = (thread_arg*)ptr;

  for(int i=0; i<arg->num; i++)
    arg->i++;

  return NULL;
}
```

- `increment()` increments each entry in a given array by one.
- consider the argument structure regarding cache size.

False Sharing: Example (2/3)

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

  pthread_t *thr = (pthread_t*) malloc( NTHREADS * sizeof(*thr) );
  thread_arg *arg = (thread_arg*) malloc( NTHREADS * sizeof(*arg) );

  for (int i=0; i<NTHREADS; i++) {
    arg[i].i = 0;
    arg[i].num = NUM * (i+1);
    pthread_create(thr + i, NULL, &increment, arg + i);
  }
  for(int i=0; i < NTHREADS; i++) {
    pthread_join(thr[i], NULL);
    printf("Value of thread %d: %d\n", i, arg[i].i);
  }
  return 0;
}
```

```
time ./false_sharing

real 0m10.202s
user 0m17.285s
sys  0m0.008s
```

False Sharing: Example (3/3)

- Problem:
  - `sizeof(thread_arg)` is less than size of chache-line
  - `malloc` allocates the structure instances subsequently
- 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;
```

```
time ./false_sharing

real 0m6.059s
user 0m9.061s
sys  0m0.000s
```

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 (1/2)

```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;
}
```

Data Hazards: Incrementing i (2/3)

```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++)
    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 (3/3)

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

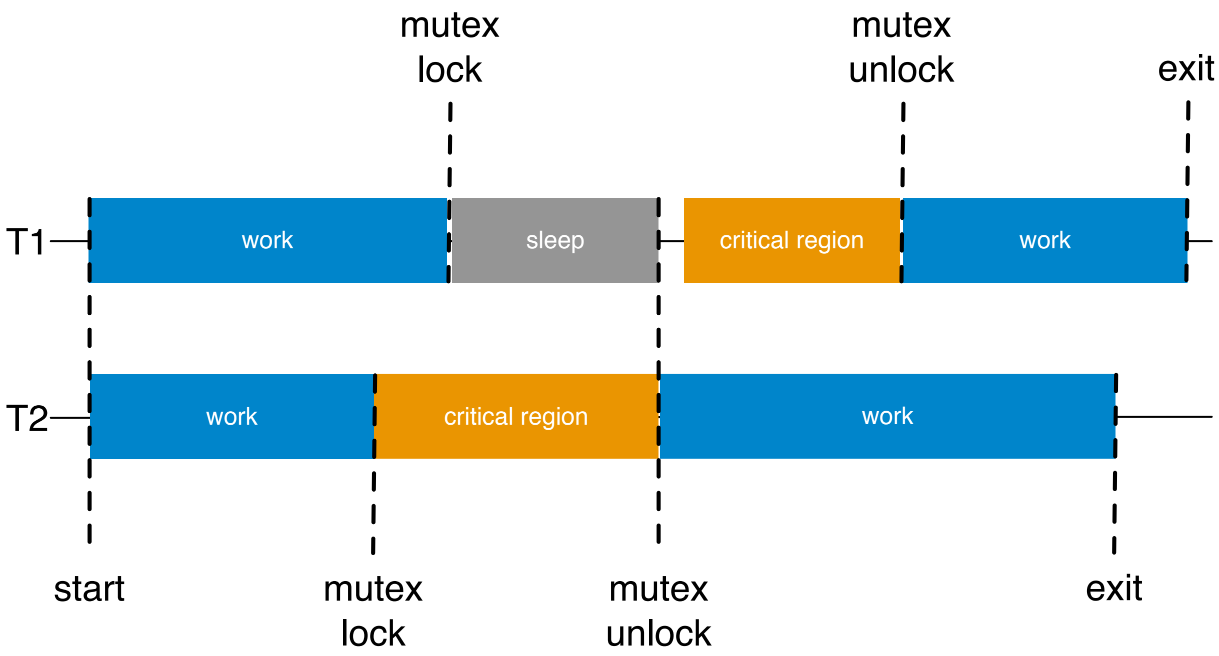
Synchronization with Pthread

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

```
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 (1/2)

```
#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 (2/2)

```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 (1/2)

```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 (2/2)

```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 (1/2)

```
#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 (2/2)

```
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
```

Mutexes: Extended Topics (1/2)

- **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 )
  ...
  ```

Mutexes: Extended Topics (2/2)

- **Semaphores**

  `int sem_init(...)`

  `int sem_post(...)`
  `int 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**

  `int barrier_init(barrier_t *barrier, int count)`

  `int barrier_destroy(barrier_t *barrier)`
  `int barrier_wait(barrier_t *barrier)`

  - `barrier_wait` waits for `count` threads until they can pass it.

Assignment 2 - histogram (dynamic)

Assignment 2: histogram (dynamic)

- Use POSIX threads to parallelize `get_histogram()`
- The program should follow the producer/consumer pattern:
  - There is one producer thread (the main thread) that prepares chunks of text blocks and writes it to a buffer.
  - There are `numWorker` worker threads that use the chunks as input and count the names asynchronously.
  - As soon as chunks are available on the buffer, a free worker grabs it and begins to count.
  - The results are used by the producer thread to progress.
- Consider:
  - The number of created worker threads N is checked (`numWorker` $<=$ N $<=$ 2 x `numWorker`).
  - You may have to use global storage, think about synchronization.
  - The speedup with 32 cores must be at least 16.
  - This time, we provide a new function `int getNameIndex(const char*)`, that returns the index of the name or -1 if not found.

Assignment 2: histogram (dynamic) - `get_histogram()`

```c
#include "names.h"

void get_histogram(char *buffer, int* histogram, int num_threads)
{
   char current_word[20] = "";
   int c = 0;

   for (int i=0; buffer[i]!=TERMINATOR; i++) {

      if(isalpha(buffer[i]) && i%CHUNKSIZE!=0 ){
         current_word[c++] = buffer[i];
      } else {
         current_word[c] = '\0';
         int res = getNameIndex(current_word); // new hash-function
         if (res != -1)
            histogram[res]++;
         c = 0;
      }

   }
}
```

Assignment: histogram (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 `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 (dynamic) - Provided Files

- names.c / .h
  - Contains `getNameIndex()` to return the name index or -1.
- 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.