# Parallel Programming Exercises


Andreas Hollmann

Technische Universität München



April 23, 2013

# Organization

- Lecture could start at:
  - 12:00
  - 12:15
  - 12:30
- Poll is available at http://doodle.com/uzy7zfg6gaatwgkg
- Duration: as long as we need, up to 90 min.
- Additional assistance on assignments:
  - Tuesday 10:00 - 12:00 room 01.04.011
  - Wednesday 14:00 - 16:00 room 01.04.011
- Detailed discussion of programming techniques and assignments in this lecture
- My email address is: hollmann@in.tum.de
- Website will be updated and extended in the next days

# Assignments

- Neither graded nor corrected
- Still very important
- Small programming tasks in final exam
  - without practice you will run out of time or do it wrong
- Solutions will be made public
- Topics
  - Pthreads (Posix Threads)
  - OpenMP (Open Multi-Processing)
  - MPI (Message Passing Interface)
- Code examples are in C99
- C++ won't be covered

# Additional Book Resources

- Students at TUM have access to Safari Books Online
  - http://proquest.tech.safaribooksonline.de.eaccess.ub.tum.de
  - Login: user@mytum.de or user@tum.de and your password
  - Online access to really good books
  - Printing single chapters is possible
  - Searching for keywords across all books
- Recommended Books
  - An Introduction to Parallel Programming, by Peter Pacheco
  - Programming with Posix Threads, by David Butenhof
  - The Linux Programming Interface, by Michael Kerrisk
  - Patterns for Parallel Programming, by Timothy G. Mattson; Beverly A. Sanders; Berna L. Massingill

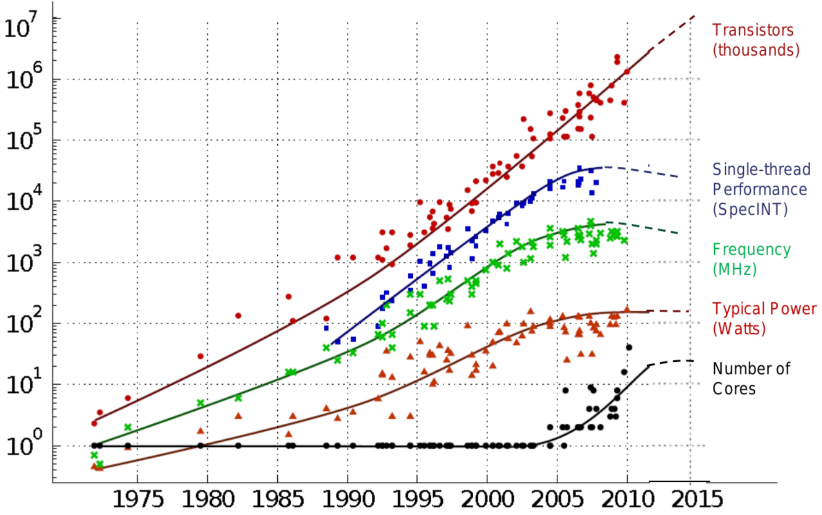
# Additional Video Resources

- ParLAB
  - Very good presentations on different parallel programming topics at University of California, Berkeley
- Ulrich Drepper
  - Scalable Parallel Programming Techniques
  - Why knowing your hardware is important

# Course Prerequisites

- Knowledge of C
  - What is a global or static variable?
  - What does const mean in C? Is 'const int *c = &b; and int const *c = &b; the same?
- C books
  - (C89) The C Programming Language, Second Edition, by Brian W. Kernighan; Dennis M. Ritchie
  - (C99) C Primer Plus, Fifth Edition, by Stephen Prata
- 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

# 35 Years of Microprocessor Trend Data




Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

# 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
- Prior to 2004, single-threaded floating-point performance climbed, at 64% per year: a doubling period of 73 weeks. After that, it leveled off at 21% per year.
- Software developers have to rewrite their applications to use multiple processors in order top speed them up
- Parallel Programming is hard
  - to write - complex APIs and needs more code than serial version
  - to do it correctly - it's easy to introduce new bugs
  - to debug, order of thread execution is undefined
  - to make it scalable - will your applications scale with more cores?
  - better qualified developers are necessary

# Is parallel programming that complex?

- It depends on the problem
- For completely independent tasks it’s rather easy
  - Embarrassingly Parallel - parameter studies or serving static files on a webserver to multiple users
- It gets more and really complex with dependencies between tasks and shared data
- Some problems are simple to understand, but really hard to implement effciently, e. g. Dijkstra’s shortest-path algorithm.
- Scientific Algorithms are ok
  - Often ends up in some stencil or matrix computation

# Posix Thread Programming

- Single process with multiple paths of execution running in parallel
  - Functions are started in new threads and will run concurrently
- Usage:
  - In GUIs: one thread updates the GUI, second thread does actual work
  - To speed up IO intensive workloads, one thread does IO, other threads do computation
  - To speed up computations, by using all CPUs
- Posix Threads were defined in 1995 (IEEE Std 1003.1c-1995)
- Threads are light-weight compared to processes
- All threads within one process share
  - Memory resources (data, heap, code area)
  - File descriptors
  - Environment (privileges, working directory)
- Each thread has a private stack

# Why are threads faster than processes?

- Creating a new process with fork() has a big overhead: whole memory must be copied
  - Waste of memory resources!
- Synchronization with processes usually involves system calls.

# Output ps tool

```
hollmann@lxhalle:~$ ps -fLmU hollmann
UID          PID    LWP  C NLWP CMD
hollmann   16233      -  0    1 sshd: hollmann@pts/29
hollmann       -  16233  0    - -
hollmann   16234      -  0    1 -bash
hollmann       -  16234  0    - -
hollmann   16279      -  0    1 sshd: hollmann@pts/31
hollmann       -  16279  0    - -
hollmann   16280      -  0    1 -bash
hollmann       -  16280  0    - -
hollmann   28347      - 99    2 ./parallel_app
hollmann       -  28347 81    - -
hollmann       -  28348 74    - -
hollmann   28349      -  0    1 ps -fLmU hollmann
hollmann       -  28349  0    - -
```

# Posix Threads on Linux

- Pthreads are mapped 1:1 to Linux threads
- A process can have many threads (at least one) 1:N and is identified by its process id (PID)
- Each linux thread has an unique thread id (TID) and belongs to exactly one process
- There is no defined printable id of a pthread, on Linux it is possible to use TID

# Hello World Example

```
int main()
{
  printf("Hello World!\n");
}
```

# Hello World with Pthreads Ver. 0

```
void * hello()
{
  printf("Hello World from pthread!\n");
  return NULL;
}

void main()
{
  pthread_t thread;

  pthread_create(&thread, NULL, &hello, NULL);

  printf("Hello World from main!\n");

  pthread_join(thread, NULL);
}
```

# Create Pthreads

```
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

```c
int pthread_join(pthread_t thread,
                 void **retval);
```

- pthread_t thread,
  - Pointer to 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.

# Compile & Output

```
gcc --std=gnu99 -pthread -Wall
    -o hello_world hello_world.c

Hello World from main!
Hello World from pthread!
```

# Hello World with Pthreads Ver. 1

```
void main()
{
  long num_threads = 3; pthread_t *thread;

  thread = malloc((num_threads * sizeof(*thread));

  for (int i = 0; i < num_threads; i++)
    pthread_create(thread + i, NULL, &hello, NULL );

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

# Output

```
[user]$ ./hello_world_pthread_1
Hello World from pthread!
Hello World from pthread!
Hello World from pthread!
```

# Hello World with Pthreads Ver. 2

```c
void * hello(void *ptr)
{
  printf("Hello World from pthread %d!\n", *((int*)ptr));
  return NULL ;
}
```

# Hello World with Pthreads Ver. 2

```
void main()
{
  int num_threads = 3;  pthread_t *thread;  int *thread_arg;

  thread = malloc(num_threads * sizeof(*thread));
  thread_arg = malloc(num_threads * sizeof(*thread_arg));

  for (int i = 0; i < num_threads; i++)
  {
    thread_arg[i] = i;
    pthread_create(thread + i, NULL, &hello, thread_arg + i);
  }

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

# Output

```
[user]$ ./hello_world_pthread_2
Hello World from pthread 0!
Hello World from pthread 1!
Hello World from pthread 2!
```

# Hello World with Pthreads Ver. 3

```c

pid_t gettid() { return (pid_t) syscall( __NR_gettid ); }

struct pthread_args
{
  long thread_id;
  long num_threads;
};

void * hello(void *ptr) {
  struct pthread_args *arg = ptr;
  printf("Hello World from pthread %ld of %ld PID = %d TID = %
          arg->thread_id,
          arg->num_threads,
          getpid(),
          gettid());

  return NULL ;
}
```

# Hello World with Pthreads Ver. 3

```c
void main()
{
  long num_threads = 3;
  pthread_t *thread;
  struct pthread_args *thread_arg;

  thread = malloc(num_threads * sizeof(*thread));
  thread_arg = malloc(num_threads * sizeof(*thread_arg));

  for (int i = 0; i < num_threads; i++)
  {
    thread_arg[i].thread_id = i;
    thread_arg[i].num_threads = num_threads;
    pthread_create(thread + i, NULL, &hello_pthread, thread_arg
  }

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

# Output

```
[user]$ ./hello_world_pthread_3
Hello World from pthread 1 of 3 PID = 23750 TID = 23752!
Hello World from pthread 0 of 3 PID = 23750 TID = 23751!
Hello World from pthread 2 of 3 PID = 23750 TID = 23753!
```

# Assignment: Numerical integration of PI

$$\pi = 4 \cdot \int_0^1 \sqrt{1 - x^2}\, dx$$

# Assignment: Parallelize this Code

```
#include <stdio.h>
#include <math.h>
#define STEPS 1000000

void main()
{
  double step_size = 1.0/STEPS, t = 0.5 * step_size, sum = 0;

  while(t < 1.0)
  {
    sum += sqrt(1-t*t) * step_size;
    t += step_size;
  }
  sum *= 4;

  printf("Computed PI = %.10lf\n", sum);
  printf("Difference to Reference is %.10lf\n", M_PI - sum);
}
```