Parallel Programming Exercises

Andreas Hollmann

Technische Universität München

April 23, 2013

Organization

- This course will start at 12:00
- No groups, no lab courses
- Assistance sessions as announced last time: on Tuesdays between 10:00 - 12:00 and on Wednesdays from 14:00 - 16:00, starting next week

Assignment: Parallelize this Code

```c
#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);
}
```

One PI Solution 1/4

```c
#define STEPS 1000000
#define STEP_SIZE 1.0/STEPS
#define THREADS 3

struct pthread_args
{
  double lower;
  double upper;
  double local_sum;
};
```

One PI Solution 2/4

```
void * pi_thread(void *ptr)
{
  double low = 0.5 * STEP_SIZE +
                  ((struct pthread_args*)ptr)->lower;
  double upp = ((struct pthread_args*)ptr)->upper;
  double tsum = 0;

  while(low < upp)
  {
    tsum += sqrt(1-low*low) * STEP_SIZE;
    low += STEP_SIZE;
  }
  ((struct pthread_args*)ptr)->local_sum = tsum;

  return NULL;
}
```

One PI Solution 3/4

```
void main()
{
  long num_threads = 10000; double sum = 0;
  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].lower = (i+0) * (1.0/(double)num_threads);
    thread_arg[i].upper = (i+1) * (1.0/(double)num_threads);
    pthread_create(thread+i, NULL, &pi_thread, thread_arg+i);
  }
  for (int i = 0; i < num_threads; i++)
  {
    pthread_join(thread[i], NULL);
    sum += 4 * thread_arg[i].local_sum;
  }
}
```

One PI Solution 3/4

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

Output PI

```
$ time ./pi
Reference PI = 3.1415926536 Computed PI = 3.1415926539
Difference to Reference is -0.0000000003

real    0m0.021s
user    0m0.020s
sys     0m0.000s

$ time ./pi_pthread_0
Reference PI = 3.1415926536 Computed PI = 3.1415926539
Difference to Reference is -0.0000000003

real    0m0.487s
user    0m0.127s
sys     0m0.660s
```

Why knowing your hardware is important?

- In order to set the right number of threads.
- To run the threads on the right CPUs

CPU Information 1/4

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 37
model name      : Intel(R) Core(TM) i7 CPU M 620  @ 2.67G
stepping        : 2
microcode       : 0xd
cpu MHz         : 1199.000
cache size      : 4096 KB
physical id     : 0
siblings        : 4
core id         : 0
cpu cores       : 2
apicid          : 0
```

CPU Information 2/4

```
processor       : 1
vendor_id       : GenuineIntel
cpu family      : 6
model           : 37
model name      : Intel(R) Core(TM) i7 CPU M 620  @ 2.67G
stepping        : 2
microcode       : 0xd
cpu MHz         : 1199.000
cache size      : 4096 KB
physical id     : 0
siblings        : 4
core id         : 2
cpu cores       : 2
apicid          : 4
```

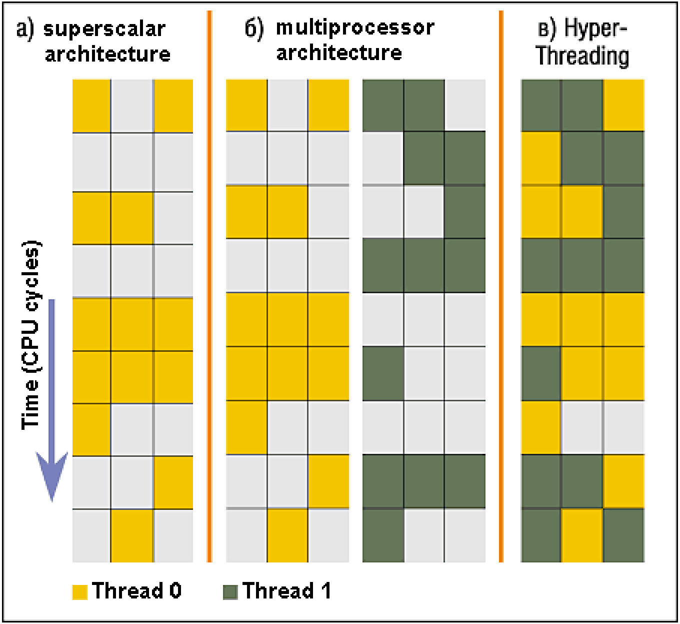
CPU Information 3/4

```
processor       : 2
vendor_id       : GenuineIntel
cpu family      : 6
model           : 37
model name      : Intel(R) Core(TM) i7 CPU M 620  @ 2.67G
stepping        : 2
microcode       : 0xd
cpu MHz         : 1199.000
cache size      : 4096 KB
physical id     : 0
siblings        : 4
core id         : 0
cpu cores       : 2
apicid          : 1
```

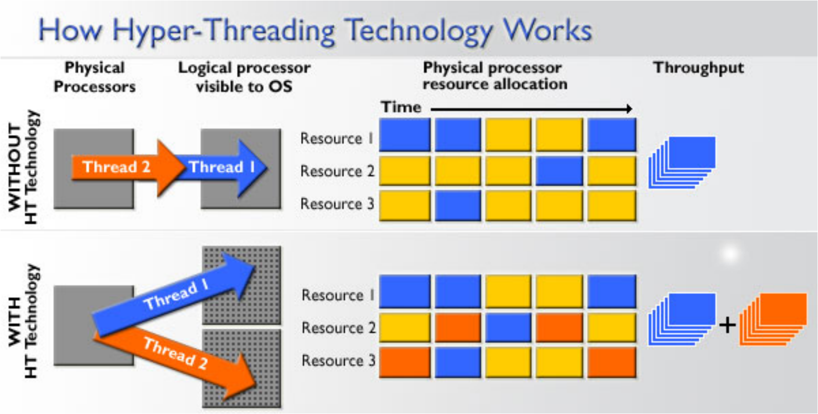
CPU Information 4/4

```
processor       : 3
vendor_id       : GenuineIntel
cpu family      : 6
model           : 37
model name      : Intel(R) Core(TM) i7 CPU M 620  @ 2.67G
stepping        : 2
microcode       : 0xd
cpu MHz         : 1199.000
cache size      : 4096 KB
physical id     : 0
siblings        : 4
core id         : 2
cpu cores       : 2
apicid          : 5
```

Hyperthreading (SMT)



Hyperthreading (SMT)



Hyperthreading (SMT)

- Superscalar architectures have several functional units that can work in parallel
- Functionl units are not well utilized with only one thread (stream of instruction)
- SMT duplicates part of a real CPU core, but shares functional units with other threads
- Improves the utilization of the functional units for certain applications or combinations of applications
  - One application does integer computation the other does floating point calculation

PI Measurements with 4 and 2 Threads

```
$ time ./pi_pthread_0 \begin
Reference PI = 3.1415926536 Computed PI = 3.1415926623
Difference to Reference is -0.0000000087 <-- compare

real    0m8.865s
user    0m34.323s <-- 4 threads
sys     0m0.027s

$ time ./pi_pthread_0
Reference PI = 3.1415926536 Computed PI = 3.1415926626
Difference to Reference is -0.0000000090 <-- differs

real    0m8.867s
user    0m17.673s <-- 2 threads
sys     0m0.000s
```

Limiting the CPU_SET with taskset command

```
$ time taskset -c 0 ./pi_pthread_0
Reference PI = 3.1415926536 Computed PI = 3.1415926626
Difference to Reference is -0.0000000090

real    0m16.568s
user    0m16.513s
sys     0m0.007s

$ time taskset -c 0,2 ./pi_pthread_0
Reference PI = 3.1415926536 Computed PI = 3.1415926626
Difference to Reference is -0.0000000090

real    0m16.577s
user    0m33.033s
sys     0m0.017s
```

Limiting the CPU_SET with taskset command

```
$ time taskset -c 0,2 ./pi_pthread_0
Reference PI = 3.1415926536 Computed PI = 3.1415926626
Difference to Reference is -0.0000000090

real    0m16.577s
user    0m33.033s
sys     0m0.017s

$ time taskset -c 0,1 ./pi_pthread_0
Reference PI = 3.1415926536 Computed PI = 3.1415926626
Difference to Reference is -0.0000000090

real    0m8.876s
user    0m17.687s
sys     0m0.000s
```

Linux 2.6.23 (CFS) Scheduling Overview

- One run queue per (logical) CPU
- Active threads are placed in one of these queues
- Thread runs until his time slice is over or it reaches a blocking functions and waits
- Operating System tries to keep all run queues balanced and migrates threads
  - Good for most application, bad for parallel applications with frequent synchronization

Incrementing i 1/2

```c
#define NUM 10000000

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

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

    return NULL;
}
```

Incrementing i 2/2

```c
void main()
{
  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);
}
```

```
$ ./increment_integer
Value of i = 11315419
$ ./increment_integer
Value of i = 11038305
```

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”

GCC Explorer Demo

- http://gcc.godbolt.org/

Incrementing i with Mutex 1/2

```c
#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;
}
```

Incrementing i with Mutex 2/2

```
void main()
{
  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);
}
```

Incrementing i with Spinlock 1/2

```c
#define NUM 10000000

pthread_spinlock_t spinlock;

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

  for(int j=0; j < NUM; j++)
  {
    pthread_spin_lock(&spinlock);
    (*i)++;
    pthread_spin_unlock(&spinlock);
  }

  return NULL;
}
```

Incrementing i with Spinlock 2/2

```c
void main()
{
   int i = 0; pthread_t thr;

   pthread_spin_init(&spinlock, PTHREAD_PROCESS_PRIVATE);
   pthread_create(&thr, NULL, &increment, &i);

   for(int j=0; j < NUM; j++)
   {
      pthread_spin_lock(&spinlock);
      i++;
      pthread_spin_unlock(&spinlock);
   }

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

Comparison Mutex and Spinlock

```
$ time ./increment_integer_mutex
Value of i = 20000000

real    0m1.079s
user    0m0.937s <-- user space
sys     0m1.137s <-- kernel space

time ./increment_integer_spinlock
Value of i = 20000000

real    0m1.062s
user    0m2.067s <-- user space only
sys     0m0.000s
```

Synchronization in General

- Bad for performance
- Serializes application (Amdahl's law)
- Hurts scalability
- Avoid if possible
  - Duplicate data
  - Rewrite algorithm
- Chose best synchronization primitive for your task

Assignment for this week: Dynamic Work Distribution

- In the last exercise on pi, the work was split in the beginning (statically) into number of threads pieces
- Each thread was computing his part of the work and in the end the result was combined
- This time the work should be distributed dynamically during runtime (STEPS)
- Protect shared variables with the introduced synchronization primitives and measure the runtime
- Compare the runtime of the static work distribution with the dynamic one.