C++11 Multithreading

Prof. Gheith Abandah

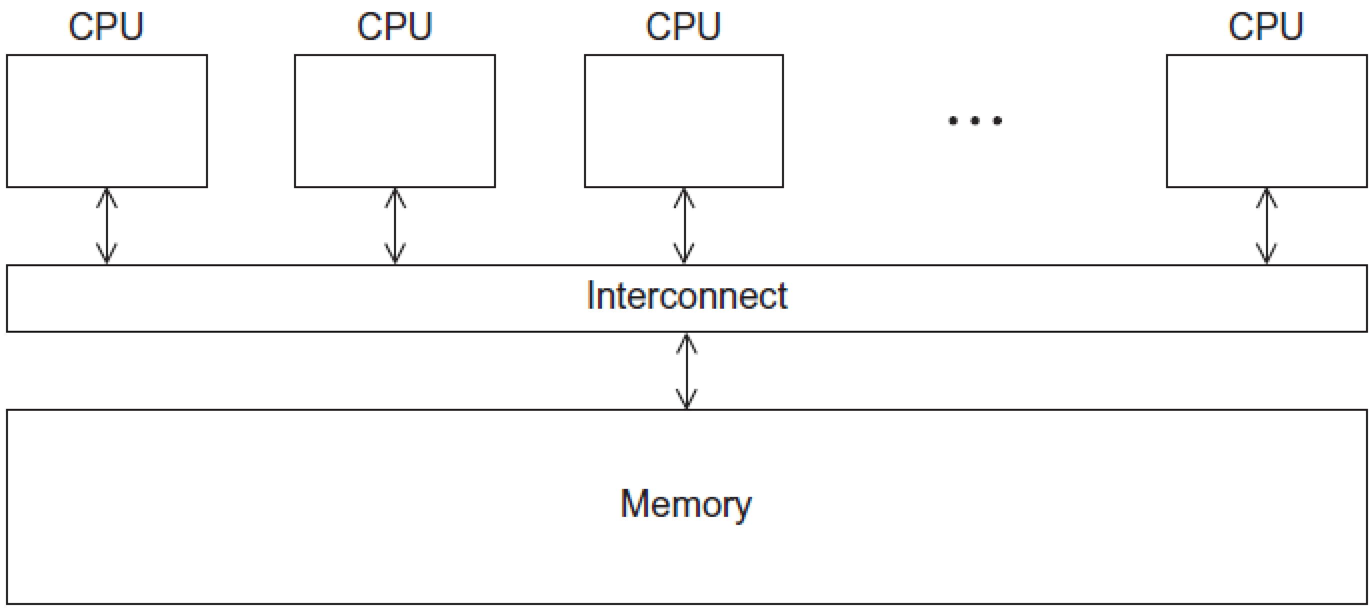
References:

- B. Schmidt, et al. Parallel Programming: Concepts and Practice. Morgan Kaufmann, 2017.
- P. Pacheco. An Introduction to Parallel Programming, Morgan Kaufmann, 2011.
- http://www.cplusplus.com/doc/tutorial/

Outline

1. Introduction
2. Handling Return Values
3. Scheduling Based on Static Distributions
4. Handling Load Imbalance
5. Signaling Threads with Condition Variables
6. Homework

1. Introduction

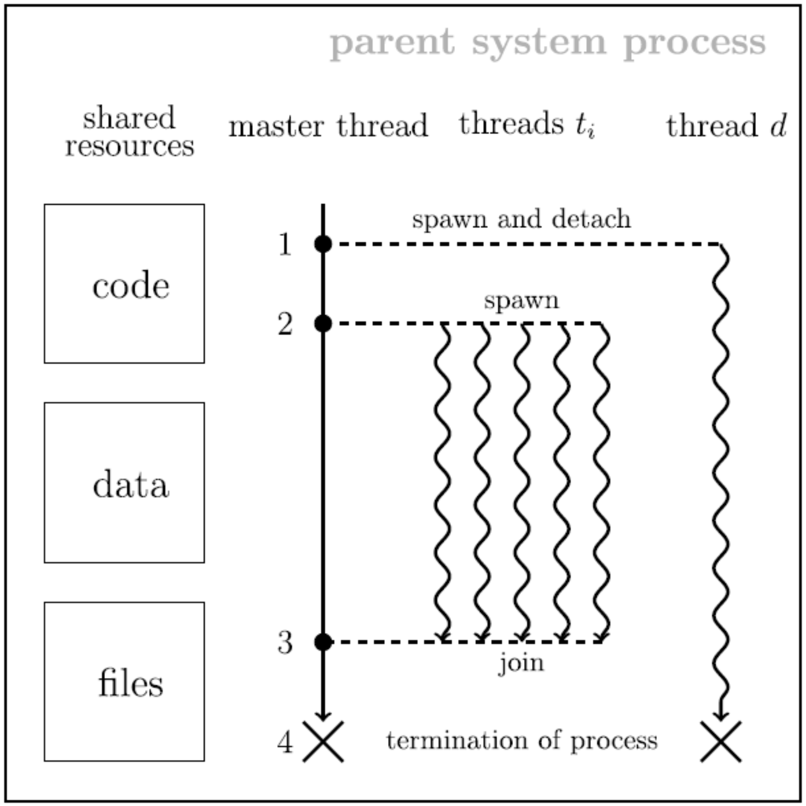


A Shared Memory System

Processes and Threads

- A process is an instance of a running (or suspended) program.
- Threads are analogous to a “light-weight” process.
- In a shared memory program, a single process may have multiple threads of control.

Spawning and Joining Threads



- An arbitrary number of software threads can be spawned by the master thread of a system process.
- Oversubscription
- **Rules:**
  1. Each thread can only be joined or detached once.
  2. A detached thread cannot be joined, and vice versa.
  3. Joined or detached threads cannot be reused.
  4. All threads have to be joined or detached within the scope of their declaration.

Multithreading APIs

- **POSIX® Threads:** Also known as Pthreads.
  - A standard for Unix-like operating systems.
  - A library that can be linked with C programs.
  - Specifies an application programming interface (API) for multi-threaded programming.
- Windows has **.NET Thread** and **Intel's Threading Building Blocks** (TBB).
- Modern C++ programming language versions (e.g., C++11 and C++14) have built in support of multithreading.

Hello World! (1)

```
#include <iostream> // std::cout
#include <cstdint>  // uint64_t
#include <vector>   // std::vector
#include <thread>   // std::thread
// this function will be called by the threads (should
      be void)
void say_hello(uint64_t id) {
   std::cout << "Hello from thread: " << id <<
      std::endl;
}
```

Hello World! (2)

```
// this runs in the master thread
int main(int argc, char * argv[]) {
  const uint64_t num_threads = 4;
  std::vector<std::thread> threads;
  for (uint64_t id = 0; id < num_threads; id++)
    // emplace the thread object in vector threads
    // call say_hello with argument id
    threads.emplace_back(say_hello, id);
  // join each thread at the end
  for (auto& thread: threads)
    thread.join();
}
```

Equivalent to:

```
threads.push_back(std::thread(say_hello, id));
```

Compiling a Pthread program

```
g++ -O2 -std=c++11 -pthread hello_world.cpp -o hello_world
```

Link in the Pthreads library

```
$./hello_world
Hello from thread: 3
Hello from thread: 1
Hello from thread: 0
Hello from thread: 2
```

Outline

1. Introduction
2. Handling Return Values
3. Scheduling Based on Static Distributions
4. Handling Load Imbalance
5. Signaling Threads with Condition Variables
6. Homework

2. Handling Return Values – Using Pointers

```
template <typename value_t, typename index_t>
void fibo(value_t n, value_t * result) {
  // initial conditions
  value_t a_0 = 0;
  value_t a_1 = 1;

  // iteratively compute the sequence
  for (index_t index = 0; index < n; index++) {
    const value_t tmp = a_0; a_0 = a_1; a_1 += tmp;
  }
  *result = a_0;
}
```

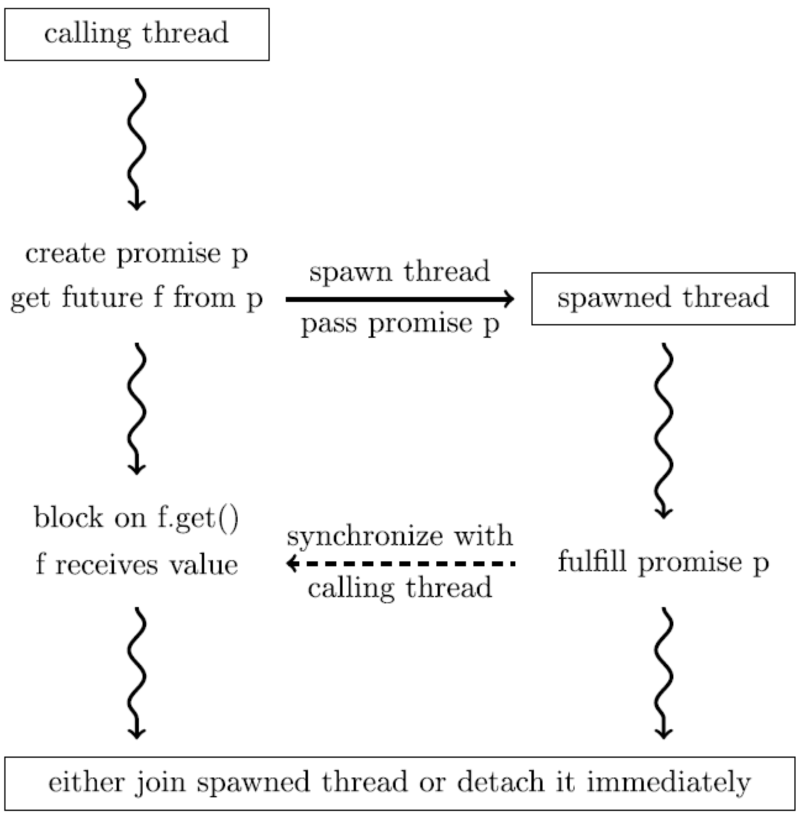
Fibonacci number: Recursively compute the n-th number using $a_n = a_{n-1} + a_{n-2}$ with initial conditions $a_0 = 0, a_1 = 1$

2. Handling Return Values – Using Pointers

```
int main(int argc, char * argv[]) {
  const uint64_t num_threads = 32;
  std::vector<std::thread> threads;
  std::vector<uint64_t> results(num_threads, 0);
  for (uint64_t id = 0; id < num_threads; id++)
    // specify template parameters and arguments
    threads.emplace_back(
        fibo<uint64_t, uint64_t>, id, &(results[id]));
  for (auto& thread: threads)
    thread.join();
  // print the result after the join
  for (const auto& result: results)
    std::cout << result << std::endl;
}
```

Address of a uint64_t

2. Handling Return Values – Using Promises and Futures



- Create the state $s = (p, f)$ by initially declaring a promise p for a specific data type T via std::promise<T> p; then assign the associated future with std::future<T> f = p.get_future();.
- The promise p is passed as rvalue reference via std::promise<T> && p. Hence, p has to be moved using std::move() from the master to the spawned thread.
- The promise p is fulfilled by setting p.set_value(some_value);.
- Read the future f using f.get(). The master thread blocks its execution until f is being signaled by p.

2. Handling Return Values – Using Promises and Futures

```
…
#include <future> // std::promise/future

template <typename value_t, typename index_t>
value_t fibo(
        value_t n,
        std::promise<value_t> && result) {

  …

  result.set_value(a_0); // <- fulfill promise
}
```

The promise, rvalue reference, no memory address

2. Handling Return Values – Using Promises and Futures

```
int main(int argc, char * argv[]) {
  …
  std::vector<std::future<uint64_t>> results;
  for (uint64_t id = 0; id < num_threads; id++) {
    // define a promise and store the associated future
    std::promise<uint64_t> promise;
    results.emplace_back(promise.get_future());
    threads.emplace_back(
      fibo<uint64_t, uint64_t>,
      id,
      std::move(promise));
  }
```

Storage for futures

Move the promise to the spawned thread. Note that promise is now moved elsewhere and cannot be accessed safely anymore.

2. Handling Return Values – Using Promises and Futures

```
  // read the futures resulting in synchronization of
     threads
  // up to the point where promises are fulfilled
  for (auto& result: results)
    std::cout << result.get() << std::endl;

  // this is mandatory since threads have to be either
  // joined or detached at the end of our program
  for (auto& thread: threads)
    thread.join();
}
```

Outline

3. Scheduling Based on Static Distributions

- For problem of size *m*, spawn *p* processors.
- Each processor computes a chunk *c*: $1 \le c \le \lceil m/p \rceil$.
  - **Block distribution**: $c = \lceil m/p \rceil$
  - **Cyclic distribution**: $c = 1$
  - **Block-cyclic distribution**: $1 < c < \lceil m/p \rceil$

global-index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
block	P_0						P_1						P_2					
cyclic	P_0	P_1	P_2	P_0	P_1	P_2	P_0	P_1	P_2	P_0	P_1	P_2	P_0	P_1	P_2	P_0	P_1	P_2
block-cyclic(2)	P_0		P_1		P_2		P_0		P_1		P_2		P_0		P_1		P_2	
block-cyclic(3)	P_0			P_1			P_2			P_0			P_1			P_2		

Example: Matrix-vector multiplication

A			
a_{00}	a_{01}	…	$a_{0,n\text{-}1}$
a_{10}	a_{11}	…	$a_{1,n\text{-}1}$
:	:		:
$a_{m\text{-}1,0}$	$a_{m\text{-}1,1}$	…	$a_{m\text{-}1,n\text{-}1}$

$\times$

$\boldsymbol{x}$
x_0
:
$x_{n\text{-}1}$

$=$

$\boldsymbol{y}$
y_0
y_1
:
$y_{m\text{-}1}$

Block Distributions (1/3)

```
template <
        typename value_t,
        typename index_t>
void block_parallel_mult(
  std::vector<value_t>& A,
  std::vector<value_t>& x,
  std::vector<value_t>& y,
  index_t m,
  index_t n,
  index_t num_threads=8) {
```

Block Distributions (2/3)

```
// inline function called by the threads that
    captures the whole scope of the reference
auto block = [&] (const index_t& id) -> void {
  const index_t chunk = m / num_threads;
  const index_t lower = id*chunk;
  const index_t upper = std::min(lower+chunk, m);

  for (index_t row = lower; row < upper; row++) {
    value_t accum = value_t(0);
    for (index_t col = 0; col < n; col++)
      accum += A[row*n+col]*x[col];
    y[row] = accum;
  }
};
```

$c = \lceil m/p \rceil$

Block Distributions (3/3)

```
std::vector<std::thread> threads;

for (index_t id = 0; id < num_threads; id++)
  threads.emplace_back(block, id);

for (auto& thread : threads)
  thread.join();
}
```

m = n = 2^{15}
T(1) = 1.29 sec, T(8) = 0.23 sec
Speedup = 5.6
Efficiency = 70%

Cyclic Distribution

```
auto cyclic = [&] (const index_t& id) -> void {

  for (index_t row = id; row < m; row += num_threads) {
    value_t accum = value_t(0);
    for (index_t col = 0; col < n; col++)
      accum += A[row*n+col]*x[col];
    y[row] = accum;
  }
};
```

c = 1

Also T(8) ≈ 0.23 sec
Simple, but may suffer from false sharing.

Block Cyclic Distribution

```
index_t chunk_size = 64/sizeof(value_t)) {
auto block_cyclic = [&] (const index_t& id) -> void {
    const index_t offset = id*chunk_size;
    const index_t stride = num_threads*chunk_size;
    for (index_t lower=offset; lower<m; lower += stride) {
      const index_t upper = std::min(lower+chunk_size, m);
      for (index_t row = lower; row < upper; row++) {
        value_t accum = value_t(0);
        for (index_t col = 0; col < n; col++)
          accum += A[row*n+col]*x[col];
        y[row] = accum;
      }
    }
  };
```

$1 < c < \lceil m/p \rceil$

Similar speedup for this example.
Most complex, but can avoid false sharing.

Outline

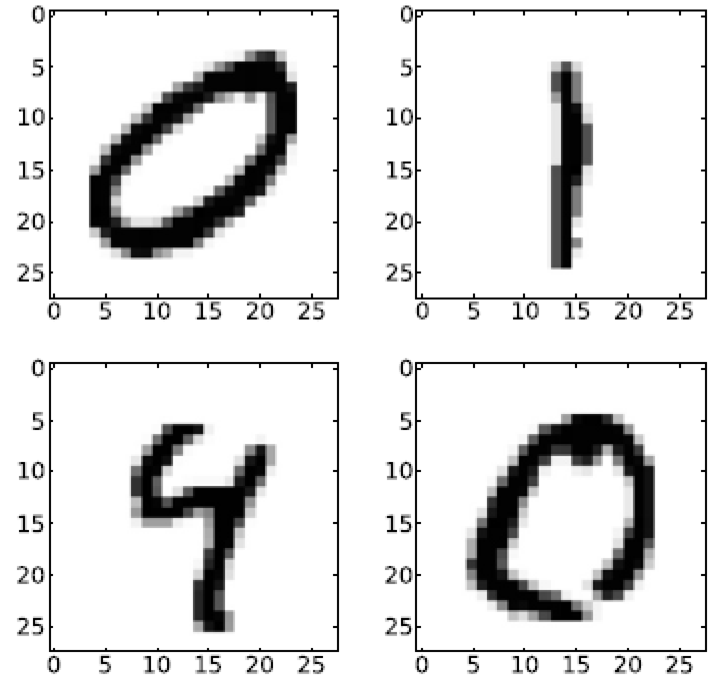
1. Introduction
2. Handling Return Values
3. Scheduling Based on Static Distributions
4. Handling Load Imbalance
5. Signaling Threads with Condition Variables
6. Homework

4. Handling Load Imbalance

- The case where a few threads still process their corresponding chunk of tasks while others have already finished their computation is called **load imbalance**.
- Solutions:
    1. Static Schedules
    2. Dynamic Block-Cyclic Distributions

Example: All-Pairs Distance Matrix

- Compute the **all-pairs distance matrix**.
- **MNIST** consists of m = 65,000 handwritten digits stored as gray-scale images of shape 28×28.
- Each image is stored as plain vector with n = 784 intensity values.



Example: All-Pairs Distance Matrix

- One used distance measure is the **squared Euclidean distance**.

$$d\left(x^{(i)}, x^{(i')}\right) := \|x^{(i)} - x^{(i')}\|^2 = \sum_{j=0}^{n-1} \left(D_{ij} - D_{i'j}\right)^2$$

- As $d\,(x^{(i)}, x^{(i')}) = d\,(x^{(i')}, x^{(i)})$, we only need to compute the lower triangular part of the distance matrix.

- Complexity: $O\,(m^2 \cdot n)$

Dist	A	B	C	D	E	F
A						
B	0.71					
C	5.66	4.95				
D	3.61	2.92	2.24			
E	4.24	3.54	1.41	1.00		
F	3.20	2.50	2.50	0.50	1.12	

Sequential Computation of All-pairs Distance Matrix

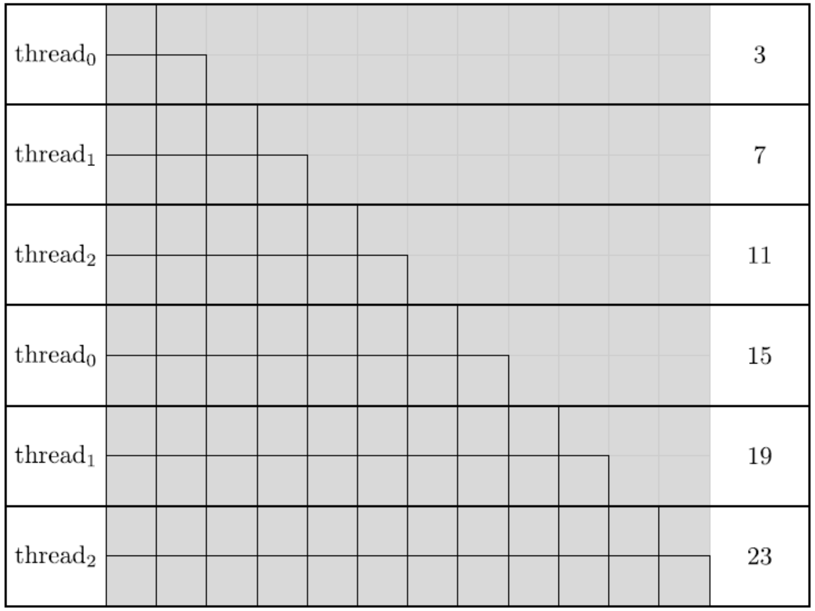
```
void sequential_all_pairs( std::vector<value_t>& mnist,
      std::vector<value_t>& all_pair,
      index_t rows, index_t cols) {
for (index_t i = 0; i < rows; i++) {
    for (index_t I = 0; I <= i; I++) {

    value_t accum = value_t(0);
    for (index_t j = 0; j < cols; j++) {
      value_t residue = mnist[i*cols+j] - mnist[I*cols+j];
      accum += residue * residue;
    }
    all_pair[i*rows+I] = all_pair[I*rows+i] = accum;
}
```

~ 30 minutes

Static Schedules

- **Block distribution** is not suitable because it takes $T(i) = \alpha \cdot (i + 1)$ to compute row i.
- **Block-cyclic distribution** is better scheme.



Block-cyclic distribution (1/2)

```
void parallel_all_pairs( std::vector<value_t>& mnist,
      std::vector<value_t>& all_pair,
      index_t rows, index_t cols,
      index_t num_threads = 64,
      index_t chunk_size = 16) {
  auto block_cyclic = [&] (const index_t& id) -> void {
  . . .}
  std::vector<std::thread> threads;
  for (index_t id = 0; id < num_threads; id++)
    threads.emplace_back(block_cyclic, id);
  for (auto& thread : threads)
    thread.join();
}
```

Block-cyclic distribution (2/2)

```
auto block_cyclic = [&] (const index_t& id) -> void {
  const index_t off = id*chunk_size;
  const index_t str = num_threads*chunk_size;
  for (index_t lower = off; lower < rows; lower += str) {
    const index_t upper = std::min(lower+chunk_size,rows);
    for (index_t i = lower; i < upper; i++) {
      for (index_t I = 0; I <= i; I++) {
        value_t accum = value_t(0);
        for (index_t j = 0; j < cols; j++) {
          value_t r = mnist[i*cols+j] - mnist[I*cols+j];
          accum += r * r; }
        all_pair[i*rows+I] = all_pair[I*rows+i] = accum;
}}}};
```

Block-cyclic distribution

- An increased chunk size causes a higher level of load imbalance resulting in longer overall execution times.

Chunk size *c*	**1**	**4**	**16**	**64**	**256**	**1024**
Time in s	44.6	45.0	45.6	49.9	57.0	78.5
Speedup	40.5	40.0	39.5	36.1	31.6	22.9

Dynamic Schedules

- Process a, b, A, B using two processors assuming $T(A) = T(B) = 10 \cdot T(a) = 10 \cdot T(b) = 10$ s.
- An optimal schedule assign the tasks {A, a} to thread 0 and {B, b} to thread 1 resulting in an overall parallel runtime of 11 seconds.
- A worst-case of {a, b} and {A, B} takes 20 seconds to compute.
- A **greedy on-demand assignment** strategy cannot be worse than 12 seconds.

Dynamic Schedules

- **Dynamic scheduling** is better than static when the computation time is unknown.
- The following code refines the static block-cyclic approach to dynamically select chunks of rows until exhausting all rows.
- A globally accessible variable **global_lower** denotes the first row of the current chunk.
- Whenever a thread runs out of work, it reads **global_lower**, and increments it by the chunk size *c*.
- The variable **global_lower** should be protected from race conditions.

```
#include <mutex>
std::mutex mutex;


mutex.lock()
// this region is only
       processed  by one
       thread at a time
mutex.unlock();
```

Dynamic block-cyclic distribution

```
std::mutex mutex;
index_t global_lower = 0;
auto dynamic_block_cyc = [&] (const index_t& id) -> void {
  index_t lower = 0;
  while (lower < rows) {
    {  // update lower row with global lower row
      std::lock_guard<std::mutex> lock_guard(mutex);
      lower = global_lower;
      global_lower += chunk_size;
    } // here the lock is released
    const index_t upper = std::min(lower+chunk_size,rows);
    for (index_t i = lower; i < upper; i++) {
      . . .
```

lock_guard locks the scope and automatically releases it on leave.

Dynamic block-cyclic distribution

- The dynamic assignment of chunks to threads is beneficial for all chunk size configurations.
- Moreover, small chunk sizes are favorable when processing tasks with heavily skewed load distributions.

Mode	Chunk size *c*	1	4	16	64	256	1024
Static	Time in s	44.6	45.0	45.6	49.9	57.0	78.5
	Speedup	40.5	40.0	39.5	36.1	31.6	22.9
Dynamic	Time in s	43.6	43.6	43.9	46.3	53.8	77.6
	Speedup	41.3	41.3	41.0	38.9	33.5	23.2

Outline

1. Introduction
2. Handling Return Values
3. Scheduling Based on Static Distributions
4. Handling Load Imbalance
5. Signaling Threads with Condition Variables
6. Homework

5. Signaling Threads with Condition Variables

- The previous **race-to-sleep** strategy fully utilizes all spawned threads until completion of their corresponding tasks – most efficient in terms of energy consumption.
- When a thread is waiting for an event, it is best to put it to sleep and wake it on event completion.
- In C++11, we can put threads to sleep and subsequently signal them to wake up using **condition variables**.

Workflow

Signaling thread

1. The signaling thread has to acquire a mutex
2. While holding the lock, the shared state is modified and sequential work is performed
3. The lock is released
4. The actual signaling by means of the condition variable **cv** is performed using **cv.notify_one**() for one thread, or **cv.notify_all**() for all threads

Waiting thread

1. A waiting thread has to acquire a **std::unique_lock** using the mutex
2. While being locked call either **cv.wait**(), **cv.wait_for**(), or **wait_until**() using **cv**. The lock is released automatically for other threads
3. In case (i) the **cv** is notified, (ii) timeout of **cv.wait**() or **cv.wait_for**(), or (iii) a spurious wake-up occurs, the thread is awaken, and the lock is reacquired. At this point, we have to check whether the globally shared state indicates to proceed or to wait (sleep) again.

Signaling of a sleeping student

```
#include <chrono> // std::this_thread::sleep_for
#include <condition_variable> // std::condition_variable
using namespace std::chrono_literals;
int main() {
  std::mutex mutex;
  std::condition_variable cv;
  bool time_for_breakfast = false; // globally shared
  auto student = [&] ( ) -> void { // called by thread
    { // this is the scope of the lock
      std::unique_lock<std::mutex> lock(mutex);
      while (!time_for_breakfast)
        cv.wait(lock);
    } // lock is finally released
    std::cout << "Time to make some coffee!" << std::endl;
  };
```

convenient time formats (C++14 required)

Allows multiple lock and unlock

Signaling of a sleeping student

```
  // create the waiting thread and wait for 2 s
  std::thread my_thread(student);
  std::this_thread::sleep_for(2s);

  { // prepare the alarm clock
    std::lock_guard<std::mutex> lock_guard(mutex);
    time_for_breakfast = true;
  } // here the lock is released

  // ring the alarm clock
  cv.notify_one();

  // wait until breakfast is finished
  my_thread.join();
}
```

One-shot synchronization using futures and promises

```
int main() {
  // create pair (future, promise)
  std::promise<void> promise;
  auto future = promise.get_future();
  auto student = [&] ( ) -> void { // called by thread
    future.get(); // blocks until fulfilling promise
    std::cout << "Time to make coffee!" << std::endl;
  };
  std::thread my_thread(student);
  std::this_thread::sleep_for(2s);
  promise.set_value(); // ring the alarm clock
  my_thread.join();
}
```

Playing ping pong

```
int main() {
  std::mutex mutex;
  std::condition_variable cv;
  bool is_ping = true; // globally shared state
  auto ping = [&] ( ) -> void {. . .}
  auto pong = [&] ( ) -> void {. . .}
  std::thread ping_thread(ping);
  std::thread pong_thread(pong);
  ping_thread.join();
  pong_thread.join();
}
```

Ping

```
auto ping = [&] ( ) -> void {
  while (true) {
    // wait to be signaled
    std::unique_lock<std::mutex> lock(mutex);
    cv.wait(lock,[&](){return is_ping;});
    std::this_thread::sleep_for(1s);
    std::cout << "ping" << std::endl;
    is_ping = !is_ping;
    cv.notify_one();
  }
};
```

Equivalent to:

```
while (! is_ping) {
    wait(lock);
}
```

Pong

```
auto pong = [&] ( ) -> void {
  while (true) {
    // wait to be signaled
    std::unique_lock<std::mutex> lock(mutex);
    cv.wait(lock,[&](){return !is_ping;});
    std::this_thread::sleep_for(1s);
    std::cout << "pong" << std::endl;
    is_ping = !is_ping;
    cv.notify_one();
  }
};
```

Summary

1. Introduction
2. Handling Return Values
3. Scheduling Based on Static Distributions
4. Handling Load Imbalance
5. Signaling Threads with Condition Variables
6. Homework

Homework

- From Textbook 1, Section 4.7, solve Exercises:
  - Exercise 2 (use the three static thread distribution patterns and one dynamic distribution pattern. Also report achieved speedup)
  - Exercise 4
  - Exercise 5