

# Lecture 02: Introduction to Parallel Programming

Vivek Kumar

Computer Science and Engineering

IIT Delhi

[vivekk@iiitd.ac.in](mailto:vivekk@iiitd.ac.in)

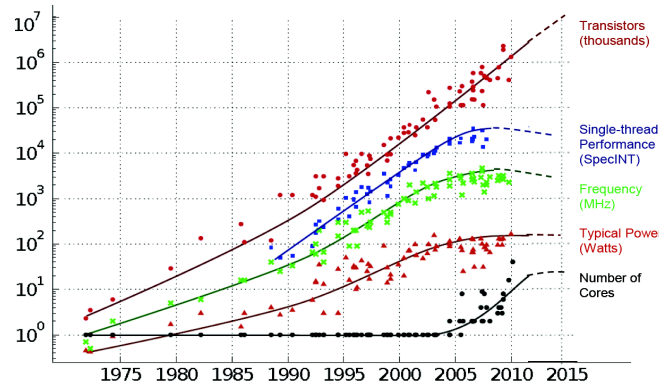
# Today's Lecture

- ➡ ● Processor technology trend
- Thread operations
- Tasks based parallel programming model
  - Functional Parallelism

Tasks-based parallel programming model and its underlying runtime system would be referred throughout in this course

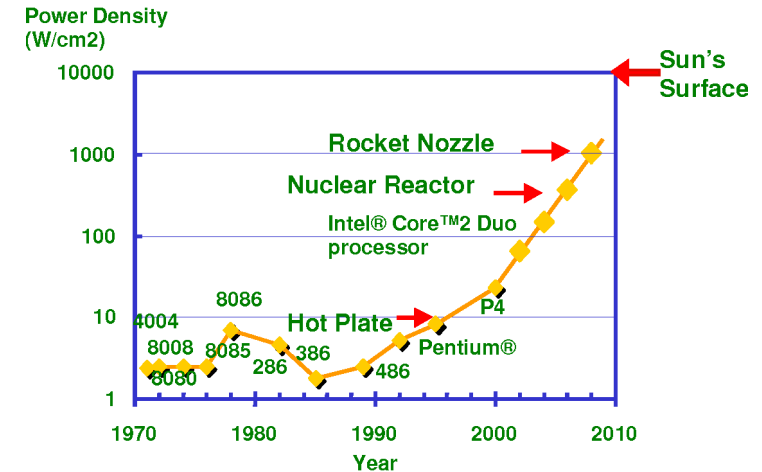
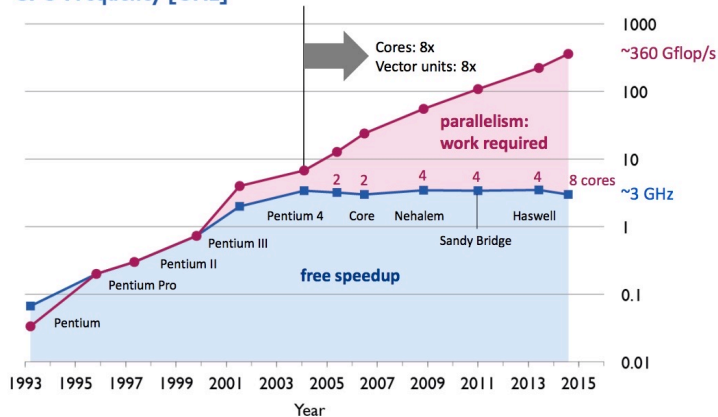
# Processor Technology Trend

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

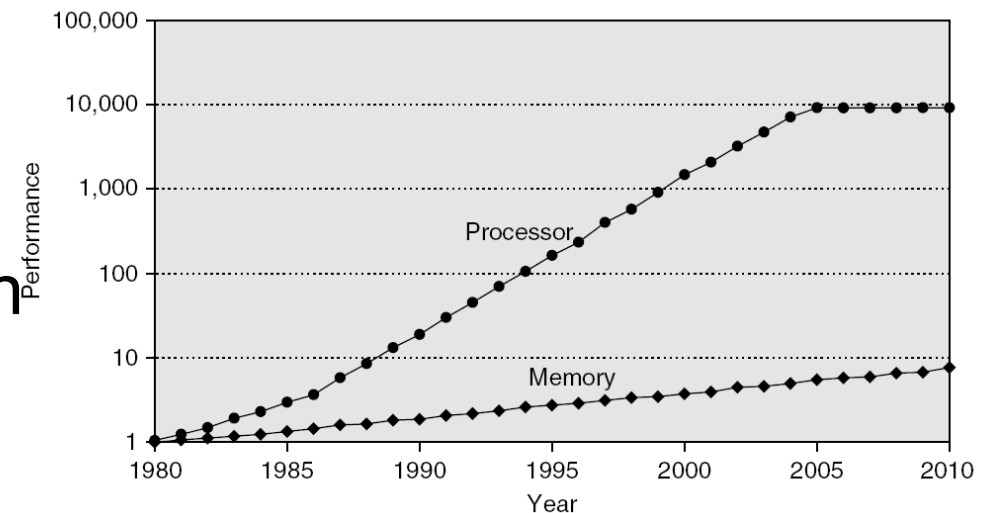
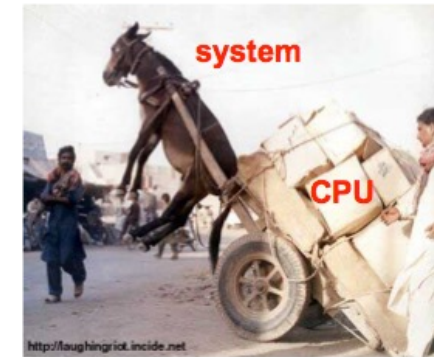
Floating point peak performance [Gflop/s]  
CPU Frequency [GHz]



- Moore's law (1964)
  - Area of transistors halves roughly every two years
    - I.e., Total transistors on processor chip gets doubled roughly every two years
- Dennard scaling (1974)
  - Power for fixed chip are remains almost constant as transistors become smaller
- No more free lunch!
  - Thermal wall hit around 2004
  - Power is proportional to cube of frequency
    - It restricts frequency growth, but opens up the multicore era

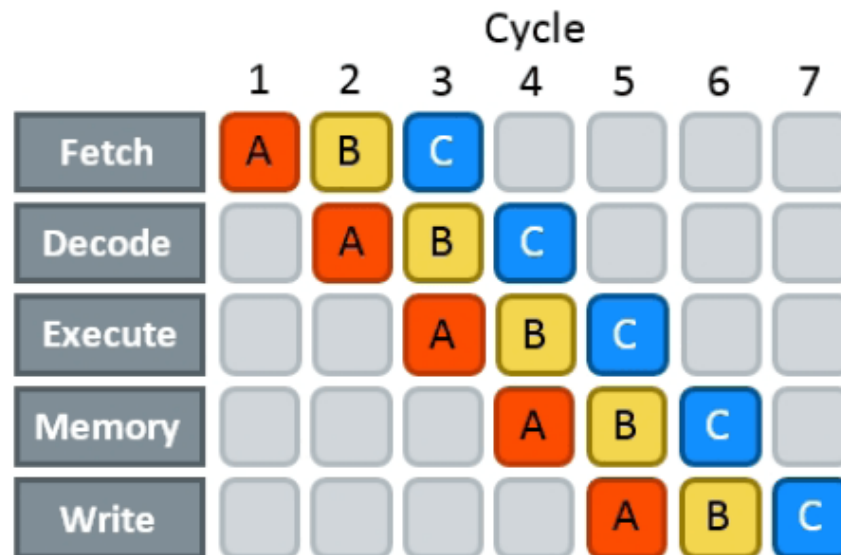
# Adding More Cores Improves performance?

- Computation is just part of the picture
- Memory latency and bandwidth
  - CPU rates have increased 4x as fast as memory over last decade
  - Bridge speed gap using memory hierarchy
  - Multicore exacerbates demand
- Inter-processor communication
- Input/Output



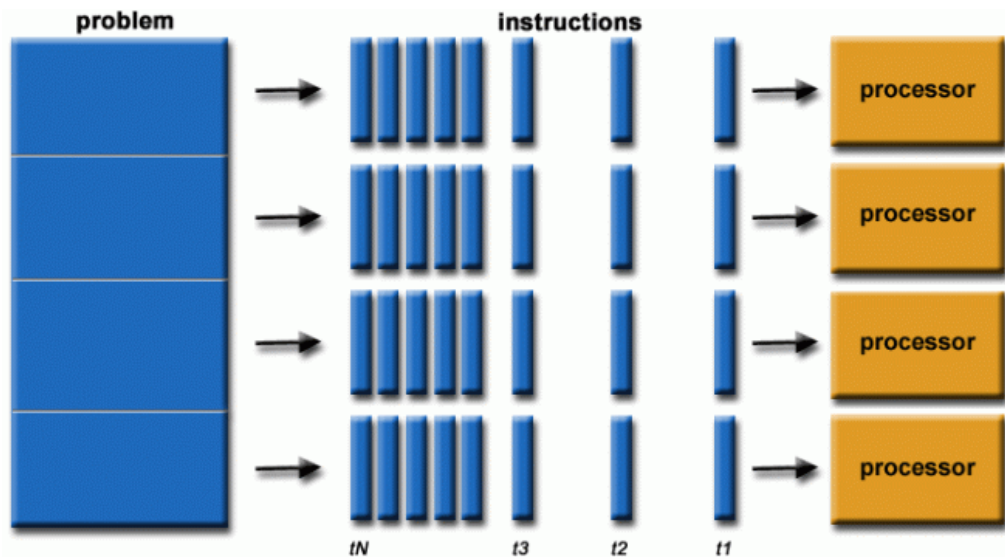
# Parallelism within a Single Core

- Instruction level parallelism (Free Parallelism)



# Free Lunch is Over!

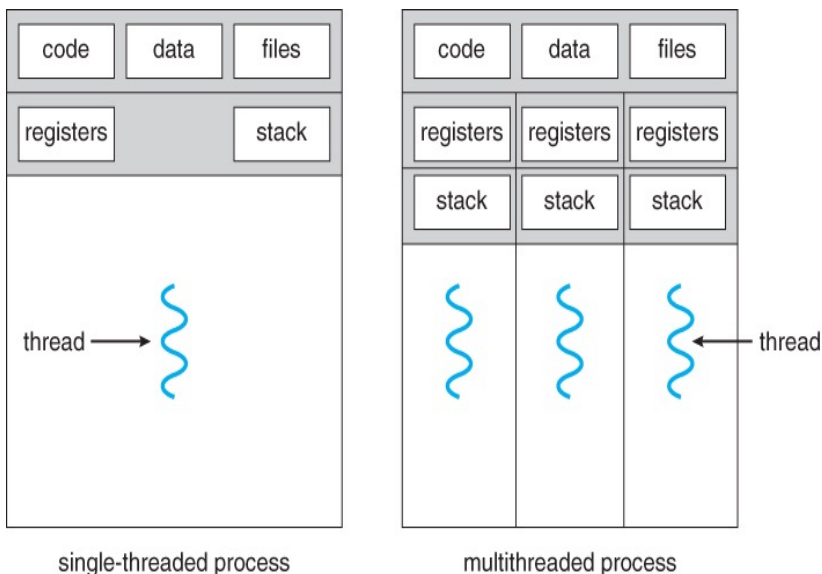
- Industrial and commercial users of parallel computing
  - BigData, Databases, Data mining
  - Artificial intelligence
  - Oil exploration
  - Web search engines, web based business services (Facebook, Twitter, etc.)
  - Medical imaging and diagnosis
  - Financial and economic modelling
  - Advanced graphics and virtual reality
  - Collaborative work environment



# Today's Lecture

- Processor technology trend
- ➔ ● Thread operations
- Tasks based parallel programming model
  - Functional Parallelism

# Thread – A Lightweight Process



- Processes are heavyweight
  - Personal address space (allocated memory)
  - Communication across process always requires help from Operating System
- Threads are lightweight
  - Share resources inside the parent process (code, data and files)
    - Easy to communicate across sibling threads!
  - They have their own personal stack (local variables, caller-callee relationship between function)
    - Each thread is assigned a different job in the program
- A process can have one or more threads

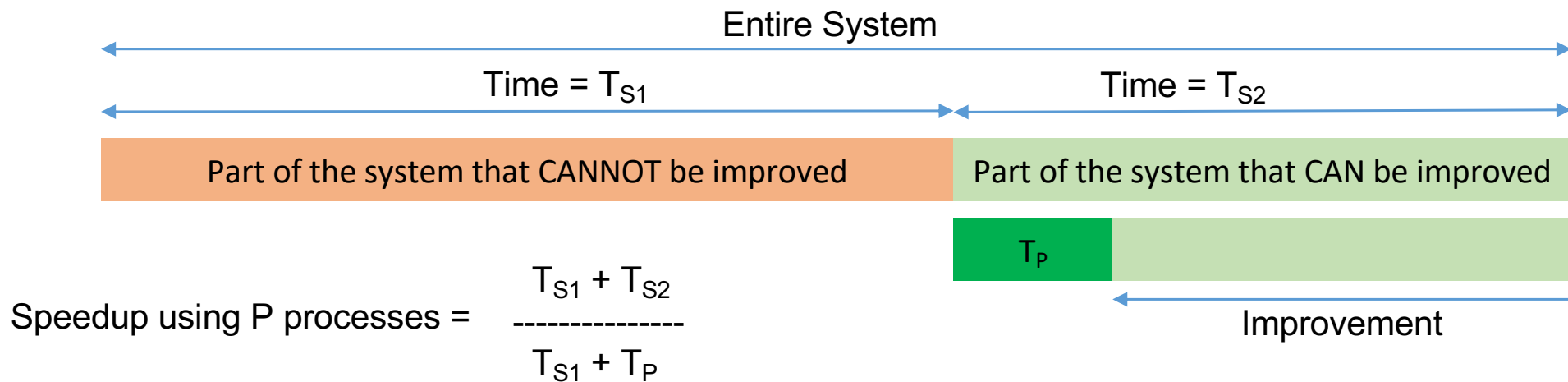


# Advantages of Multithreading

- Responsiveness
  - Even if part of program is blocked or performing lengthy operation, multithreading allows the program to continue
- Economical resource sharing
  - Threads share memory and resources of their parent process which allows multiple tasks to be performed simultaneously inside the process
- Utilization of multicores
  - Easily scale on modern multicore processors

# Amdahl's Law

- Gives an estimate of maximum expected improvement  $S$  to an overall system when only part of the system  $F_E$  is improved by a factor  $F_I$



# Thread Creation and Join Operations

- Creation

- `std::thread T(/* Pass a lambda function */ [=]() {  
    S1();  
});  
    S2();`
- Choose the correct option
  - S1 & S2 will execute in parallel
  - S1 & S2 may execute in parallel
  - S2 will execute before S1

- Join

- `T.join();`
  - It is a blocking operation, and returns once the thread **T** terminates

# Parallel Fibonacci

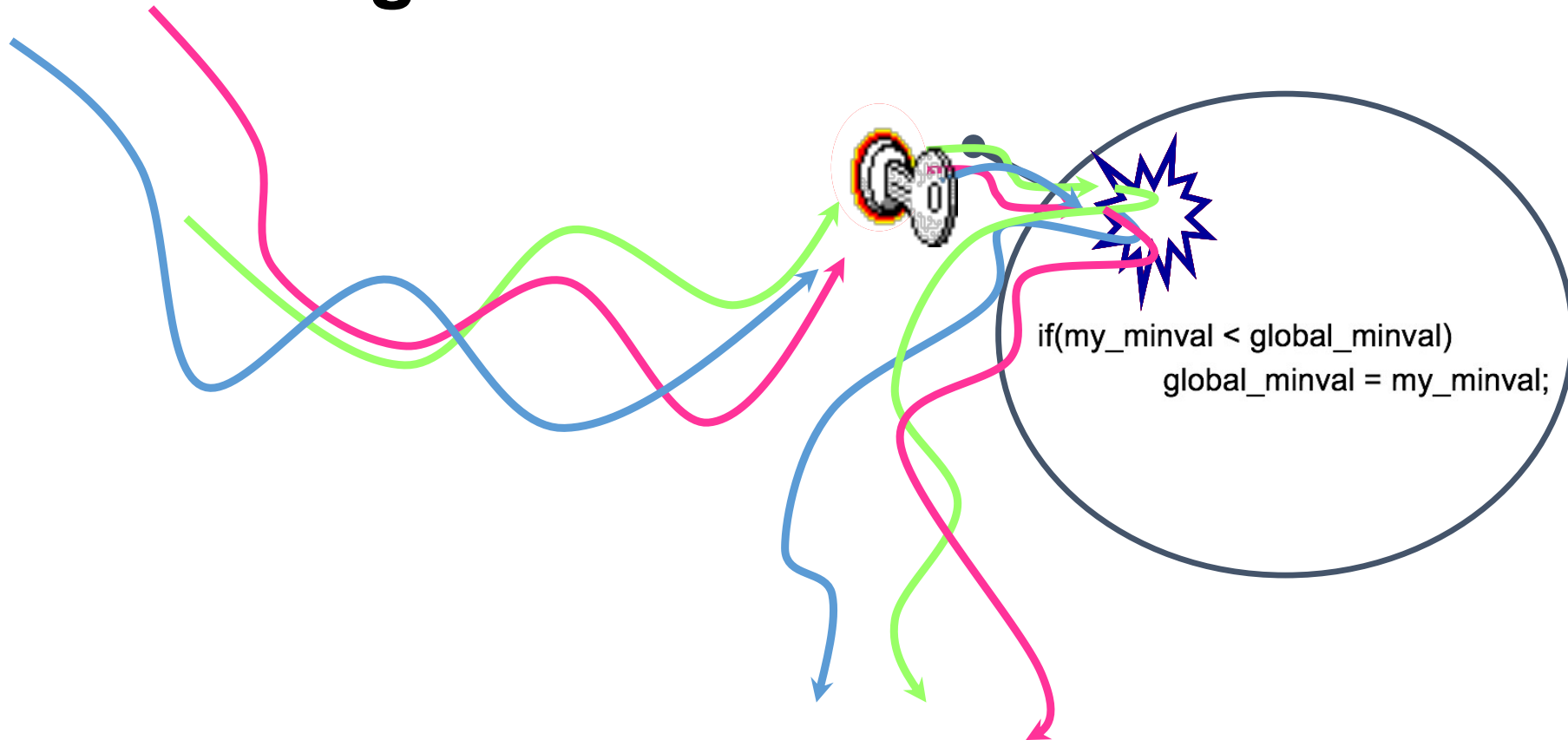
```
uint64_t fib(uint64_t n) {
    if (n < 2) {
        return n;
    } else {
        uint64_t x = fib(n-1);
        uint64_t y = fib(n-2);
        return (x + y);
    }
}
```

```
int main(int argc, char *argv[]) {
    uint64_t result;
    if (argc < 2) { return 1; }
    uint64_t n = strtoul(argv[1], NULL, 0);
    if (n < 30) {
        result = fib(n);
    } else {
        uint64_t x, y;
        std::thread T1([&]() { x = fib(n-1); });
        // main can continue executing
        y = fib(n-2);
        // wait for the thread to terminate.
        T1.join();
        result = x + y;
    }
    cout<<"Fibonacci of "<<n<<" is "<<result<<endl;
    return 0;
}
```

# Critical Section and Mutual Exclusion

- Critical section is the code executed by only one thread at a time  
/\*threads compete for update\*/  
if(my\_minval < global\_minval)  
    global\_minval = my\_minval;
- Mutex locks enforce mutual exclusion in threads as only one thread can local a mutex at any particular time
- Using mutex lock
  - request lock before executing critical section
  - enter critical section when lock granted
  - release lock when leaving critical section
- Operations  
std::mutex \_mtx;  
std::unique\_lock lck(\_mtx); /\*performs lock/unlock operation in current scope\*/

# Visualizing Critical Section & Mutual Exclusion



# Reduction using Mutex Lock

```
std::mutex _mtx;  
int minval;  
...  
void *find_minval(void *list_ptr) {  
    ...  
    std::unique_lock lck(_mtx);  
    /* mutex is automatically locked */  
    if (my_minval < minval)  
        minval = my_minval;  
    /* mutex is automatically unlocked */  
}
```

Critical Section

# Condition Variables for Synchronization

- Condition variable is associated with a predicate and a mutex
- Using a condition variable
  - thread can block itself until a condition becomes true
    - thread locks a mutex
    - tests a predicate defined on a shared variable
      - if predicate is false, then wait on the condition variable
      - waiting on condition variable unlocks associated mutex
  - when some thread makes a predicate true
    - that thread can signal the condition variable to either
      - wake one waiting thread
      - wake all waiting threads
    - when thread releases the mutex, it is passed to first waiter



# Producer Consumer Problem

```
std::mutex _mtx;
```

```
std::condition_variable _cv;
```

```
bool available=false;
```

```
L1: std::unique_lock lck(_mtx);
```

```
L2: _cv.wait(lck, [ ]() { return available;});
```

```
L3: consume_data();
```

```
L4: available=false;
```

**Consumer**

```
L5: std::unique_lock lck(_mtx);
```

```
L6: if(!available) {
```

```
L7:     create_data();
```

```
L8:     available=true;
```

```
L9:     _cv.notify_one(lck);
```

```
L10: }
```

**Producer**

# Today's Lecture

- Processor technology trend
- Thread operations
- ➔ ● Tasks based parallel programming model
  - Functional Parallelism

# Using Explicit Multithreading is Ugly (C)

```

1. uint64_t fib(uint64_t n) {
2.     if (n < 2) {
3.         return n;
4.     } else {
5.         uint64_t x = fib(n-1);
6.         uint64_t y = fib(n-2);
7.         return (x + y);
8.     }
9. }
10. int main(int argc, char *argv[]) {
11.     uint64_t result = fib(40);
12.     printf("Result is %" PRIu64 ".\n", result);
13. }

```

**Pthreads**



```

1. uint64_t fib(uint64_t n) {
2.     if (n < 2) {
3.         return n;
4.     } else {
5.         uint64_t x = fib(n-1);
6.         uint64_t y = fib(n-2);
7.         return (x + y);
8.     }
9. }
10. typedef struct {
11.     uint64_t input;
12.     uint64_t output;
13. } thread_args;
14. void *thread_func(void *ptr) {
15.     uint64_t i = ((thread_args *) ptr)-
16.         >input;
17.     ((thread_args *) ptr)->output = fib(i);
18.     return NULL;
19. }
20. int main(int argc, char *argv[]) {
21.     pthread_t thread;
22.     thread_args args;
23.     int status;
24.     uint64_t result;
25.     args.input = n-1;
26.     status = pthread_create(&thread,
27.         NULL, thread_func, &args);
28.     if (status != NULL) { return 1; }
29.     result = fib(n-2);
30.     // wait for the thread to terminate.
31.     status = pthread_join(thread, NULL);
32.     if (status != NULL) { return 1; }
33.     result += args.output;
34.     printf("Result is %" PRIu64 ".\n", result);
35.     return 0;
36. }

```

## ● Issues?

- Scalability
  - This code is only for 2 cores. Rewrite for more cores
- Modularity
  - Logic no more neatly encapsulated
- Overhead
  - Recreating thread  $>10^4$  cycles

# Using Explicit Multithreading is Ugly (C++)

```

1. uint64_t fib(uint64_t n) {
2.     if (n < 2) {
3.         return n;
4.     } else {
5.         uint64_t x = fib(n-1);
6.         uint64_t y = fib(n-2);
7.         return (x + y);
8.     }
9. }
10. int main(int argc, char *argv[]) {
11.     uint64_t result = fib(40);
12.     printf("Result is %" PRIu64 ".\n", result);
13.     return 0;
14. }

```

**std::thread**



```

uint64_t fib(uint64_t n) {
    if (n < 2) {
        return n;
    } else {
        uint64_t x, y;
        std::thread t1([&]() {x = fib(n-1);});
        std::thread t2([&]() {y = fib(n-2);});
        t1.join(); t2.join();
        return (x + y);
    }
}

```

## ● Issues?

- Scalability
  - This code doesn't work for  $N > 20$  (on my server)
- ~~Modularity~~
  - ~~Logic no more neatly encapsulated~~
- Overhead
  - Recreating thread  $> 10^4$  cycles

# Tasks Based Parallel Programming Model

```
uint64_t fib(uint64_t n) {
    if (n < 2) {
        return n;
    } else {
        uint64_t x, y;
        finish([&]() {
            async([&]() { x = fib(n-1); });
            y = fib(n-2);
        });
        return (x + y);
    }
}
```

- High productivity due to **serial elision**
  - Removing all **async** and **finish** constructs results in a valid sequential program
  - Several existing frameworks support this programming model, although the name of the APIs for tasking would be different
- Uses an underlying high performance **parallel runtime** system for load balancing of dynamically created asynchronous tasks

	Java Fork/Join	Cilk	OpenMP	HClib[1]	TBB	C++11
Serial Elision	<b>NO</b>	<b>Yes</b> spawn-sync	<b>Yes</b> #pragma omp task #pragma omp taskwait	<b>Yes</b> async-finish	<b>NO</b>	<b>Yes</b> async-future
Performance	<b>Limited</b>	<b>High</b>	<b>Limited</b>	<b>High</b>	<b>High</b>	<b>NO</b>

Popular options for simple tasks based parallel programming model

[1] <http://habanero-rice.github.io/hclib/>

# Functional Parallelism using `std::async`

```
int main(int argc, char** argv) {  
    std::future<int> part1 = std::async( [=]() { // Task-T1  
        int res = DO_SOME_WORK();  
        return res;  
    });  
    int part2 = DO_SOME_OTHER_WORK();  
    //get will block until result is ready    // Task-T2  
    int total = part1.get() + part2;  
}
```

Two issues to be addressed:

- 1) Distinction between container and value in container (`future`)
- 2) Synchronization to avoid race condition in container accesses

# Parallel Fibonacci using `std::async`

```
uint64_t fib(uint64_t n) {  
    if(n<2) {  
        return n;  
    } else {  
        std::future<uint64_t> f1 = std::async( [=]() { return fib(n-1); });  
        std::future<uint64_t> f2 = std::async( [=]() { return fib(n-2); });  
        //get will block until result is ready  
        return f1.get() + f2.get();  
    }  
}
```

Let us try some demo to see its performance

# Reading Materials

- <https://doi.org/10.1007/s11227-018-2238-4>



# Next Lecture 03

- Parallel runtime systems
- Context switching inside the user space
  - Boost Fiber library
  - Argobots runtime system
  - Project deliverable-1 based on Lecture 03 & 04