香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

Parallel Programming in C++

Daniel Hsieh {QwertyPi}

2026-06-29

Outline

- Why parallel programming?
- Processes and Threads
- Data Race and Mutex
- Synchronization Methods
- Benchmarking
- Parallel Algorithms Design Tips
- Lab Exercises!

Why parallel programming?

- How to speed up processing?

- Increase hardware clock rate
- Issue: heat dissipation

- => **Run in parallel**

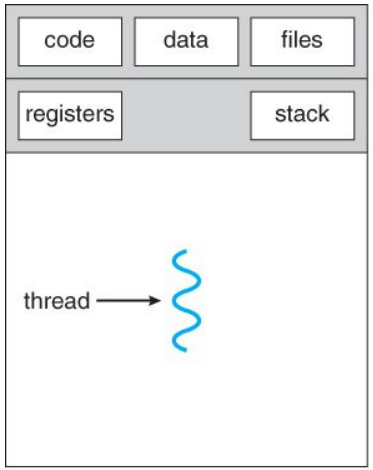
Processes and Threads

- **Process**: Execution of a program
- **Thread**: Segment of process

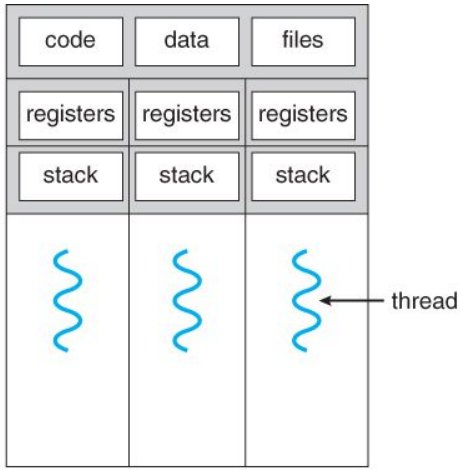
Threads are **shared memory**:

- Global (BSS) and Heap variables shared
- Register and local (stack) variables isolated

- We focus on **multi-threading** today



single-threaded process



multithreaded process

Sum of Array in Parallel

Target: Compute the sum of $n = 10^8$ integers

```
long long global_sum = 0;
for (int i = 0; i < n; i++) {
    global_sum += a[i];
}
```

```
Sum = 2147378617698223 82
Time Elapsed = 224864µs
```

This problem is ***embarrassingly parallel***:
We can split it into several unrelated tasks easily

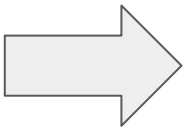
Sum of Array in Parallel

Target: Compute the sum of $n = 10^8$ integers

```cpp
long long global_sum = 0;
for (int i = 0; i < n; i++) {
    global_sum += a[i];
}
```

```
Sum = 2147378617698223 82
Time Elapsed = 224864µs
```

3.95x



```cpp
long long global_sum = 0;
mutex sum_mutex;
void compute_sum(vector<int>& a, int start, int end) {
    long long sum = 0;
    for (int i = start; i < end; i++) sum += a[i];
    lock_guard<mutex> lock(sum_mutex);
    global_sum += sum;
}
int num_threads = 4;
vector<thread> threads;
int chunk_size = (n + num_threads - 1) / num_threads;
for (int t = 0; t < num_threads; t++) {
    int start = t * chunk_size;
    int end = min(start + chunk_size, n);
    threads.emplace_back(compute_sum, ref(a), start, end);
}
for (auto& th : threads) {
    th.join();
}
```

```
Sum = 2147378617698223 82
Time Elapsed = 56925µs
```

Sum of Array in Parallel

2. “**Locks**” the global sum

1. Constructs and **spawns** the thread

3. Waits for the thread to **end and join**

```cpp
long long global_sum = 0;
mutex sum_mutex;
void compute_sum(vector<int>& a, int start, int end) {
    long long sum = 0;
    for (int i = start; i < end; i++) sum += a[i];
    lock_guard<mutex> lock(sum_mutex);
    global_sum += sum;
}
int num_threads = 4;
vector<thread> threads;
int chunk_size = (n + num_threads - 1) / num_threads;
for (int t = 0; t < num_threads; t++) {
    int start = t * chunk_size;
    int end = min(start + chunk_size, n);
    threads.emplace_back(compute_sum, ref(a), start, end);
}
for (auto& th : threads) {
    th.join();
}
```

Sum of Array in Parallel - Data Race

What if no lock,
And we directly add to `global_sum`?

```
Sum = 67425740374814124
Time Elapsed = 383729µs
```

The sum is smaller than expected...
Also, it is (comparatively) super slow

Exemplar of **data races**

```cpp
long long global_sum = 0;
// mutex sum_mutex;
void compute_sum(vector<int>& a, int start, int end) {
    // long long sum = 0;
    for (int i = start; i < end; i++) global_sum += a[i];
    // lock_guard<mutex> lock(sum_mutex);
    // global_sum += sum;
}
int num_threads = 4;
vector<thread> threads;
int chunk_size = (n + num_threads - 1) / num_threads;
for (int t = 0; t < num_threads; t++) {
    int start = t * chunk_size;
    int end = min(start + chunk_size, n);
    threads.emplace_back(compute_sum, ref(a), start, end);
}
for (auto& th : threads) {
    th.join();
}
```

Data Races

```
BSS                          global_cnt = 0;
```

```
Thread 1
int N = 100000;
for (int i = 0; i < N; i++) {
     global_cnt++;
}
```

```
Thread 2
int N = 100000;
for (int i = 0; i < N; i++) {
     global_cnt++;
}
```

```
Expect: 200000
Actual: 117963
??????
```

Data Races

Let's pull out our debugger...

register

```
eax := global_cnt
   eax := eax + 1
global_cnt := eax
```

```
gef➤  disas increment_counter
Dump of assembler code for function _Z17increment_counterv:
   0x00000000000012e9 <+0>:     endbr64
   0x00000000000012ed <+4>:     push   rbp
   0x00000000000012ee <+5>:     mov    rbp,rsp
   0x00000000000012f1 <+8>:     mov    DWORD PTR [rbp-0x4],0x18
6a0
   0x00000000000012f8 <+15>:    mov    DWORD PTR [rbp-0x8],0x0
   0x00000000000012ff <+22>:    jmp    0x1314 <_Z17increment_co
unterv+43>
   0x0000000000001301 <+24>:    mov    eax,DWORD PTR [rip+0x4e4
d]        # 0x6154 <global_cnt>
   0x0000000000001307 <+30>:    add    eax,0x1
   0x000000000000130a <+33>:    mov    DWORD PTR [rip+0x4e44],e
ax        # 0x6154 <global_cnt>
   0x0000000000001310 <+39>:    add    DWORD PTR [rbp-0x8],0x1
   0x0000000000001314 <+43>:    mov    eax,DWORD PTR [rbp-0x8]
   0x0000000000001317 <+46>:    cmp    eax,DWORD PTR [rbp-0x4]
   0x000000000000131a <+49>:    jl     0x1301 <_Z17increment_co
unterv+24>
   0x000000000000131c <+51>:    nop
   0x000000000000131d <+52>:    nop
   0x000000000000131e <+53>:    pop    rbp
   0x000000000000131f <+54>:    ret
End of assembler dump.
```

Conclusion: `global_cnt++` translates to three instructions, not one
⇒ This operation is **not atomic**.

Data Races

```
BSS                          global_cnt = 0;
```

```
Thread 1

eax := global_cnt // eax: 0
```

```
Thread 2
```

Data Races

```
BSS                          global_cnt = 0;
```

```
Thread 1

eax := global_cnt // eax: 0
```

Context switch

```
Thread 2

eax := global_cnt // eax: 0
```

Data Races

```
BSS                          global_cnt = 0;
```

```
Thread 1

eax := global_cnt // eax: 0
```

```
Thread 2
```

Context switch

```
eax := global_cnt // eax: 0
```

```
eax := eax + 1    // eax: 1
```

Data Races

```
BSS                          global_cnt = 1;
```

```
Thread 1

eax := global_cnt // eax: 0
```

```
Thread 2
```

```
eax := global_cnt // eax: 0
```

```
eax := eax + 1    // eax: 1
```

Context switch

Registers are **not** shared!

```
eax := eax + 1    // eax: 1
global_cnt := eax
```

Data Races

```
BSS                          global_cnt = 1;
```

```
Thread 1

eax := global_cnt // eax: 0
```

```
Thread 2
```

```
eax := global_cnt // eax: 0
```

```
eax := eax + 1    // eax: 1
```

Registers are **not** shared!

Context switch

```
eax := eax + 1    // eax: 1
global_cnt := eax
```

```
global_cnt := eax
```

Data Races

```
BSS                          global_cnt = 1;
```

```
Thread 1
eax := global_cnt // eax: 0

eax := eax + 1 // eax: 1


global_cnt := eax
```

```
Thread 2

eax := global_cnt // eax: 0

eax := eax + 1 // eax: 1
global_cnt := eax
```

An **instruction interleaving** where `global_cnt` has a final value of 1!

Data Races

- Data races occur when two threads:
  - Access the **same *non-atomic* memory location** simultaneously.
  - At least one of the accesses is a **write**.
  - There is **no synchronization** between the two threads.

- In C++, data races will lead to **undefined behaviour**.

Mutual Exclusion: Failed Attempt

- We have to avoid both threads simultaneously accessing `global_cnt`.
- What about adding another variable?

```
if (!in_use) {
  in_use = true;
  global_cnt++;
  in_use = false;
}
```

Mutual Exclusion: Failed Attempt

```
BSS              in_use = false;        global_cnt = 0;
```

```
Thread 1

if (!in_use)       // TRUE
```

```
Thread 2
```

Mutual Exclusion: Failed Attempt

```
BSS              in_use = false;        global_cnt = 0;
```

```
Thread 1

if (!in_use)        // TRUE
```

Context switch

```
Thread 2

if (!in_use)        // TRUE
```

Mutual Exclusion: Failed Attempt

```
BSS                in_use = false;          global_cnt = 0;
```

```
Thread 1

if (!in_use)       // TRUE
```

Context switch

```
Thread 2

if (!in_use)       // TRUE
```

```
in_use = true;
global_cnt++;
in_use = false;
```

```
in_use = true;
global_cnt++;
in_use = false;
```

Mutual Exclusion in C++

- Threads should not access the same memory at the same time
- Hence called **Mut**ual **Ex**clusion
- C++ mutex: `lock()` and `unlock()`
- Once one thread locks the mutex, the other waits for it to unlock, avoiding data race

```
mutex cnt_mutex;
int N = 100000;
for (int i = 0; i < N; i++) {
    cnt_mutex.lock();
    global_cnt++;
    cnt_mutex.unlock();
}

// global_cnt = 200000
```

Mutual Exclusion in C++

```
BSS        cnt_mutex: Locked;        global_cnt = 0;
```

```
Thread 1

cnt_mutex.lock();
```

```
Thread 2
```

Mutual Exclusion in C++

```
BSS        cnt_mutex: Locked;        global_cnt = 0;
```

```
Thread 1

cnt_mutex.lock();
```

Context switch

```
Thread 2

cnt_mutex.lock();
```

Mutual Exclusion in C++

```
BSS          cnt_mutex: Locked;          global_cnt = 0;
```

```
Thread 1

cnt_mutex.lock();
```

Context switch

```
Thread 2

cnt_mutex.lock();
```

Mutex is already locked
⇒ The thread **blocks**.

Mutual Exclusion in C++

```
BSS          cnt_mutex: Unlocked;      global_cnt = 1;
```

```
Thread 1

cnt_mutex.lock();
```

```
Thread 2
```

Context switch

```
cnt_mutex.lock();
```

```
global_cnt++;
cnt_mutex.unlock();
```

Mutual Exclusion in C++

```
BSS          cnt_mutex: Locked;          global_cnt = 1;
```

```
Thread 1

cnt_mutex.lock();
```

```
Thread 2
```

The thread **wakes up** and locks the mutex!

```
cnt_mutex.lock();
```

```
global_cnt++;
cnt_mutex.unlock();
```

Mutual Exclusion in C++

```
BSS          cnt_mutex: Unlocked;      global_cnt = 2;
```

```
Thread 1

cnt_mutex.lock();
```

```
Thread 2
```

```
cnt_mutex.lock();
```

```
global_cnt++;
cnt_mutex.unlock();
```

```
global_cnt++;
cnt_mutex.unlock();
```

RAII (Resource Acquisition Is Initialisation)

Questions:
- Why constructor spawns thread?
- Why don't we need to "unlock"?

- Ties **resources** to **object lifetime**
- **Constructor: acquire resources**
- **Destructor: release resources**

- C++ does RAII **by default** for *automatic* storage (stack)
- No need manually manage memory that much anymore!

```
long long global_sum = 0;
mutex sum_mutex;
void compute_sum(vector<int>& a, int start, int end) {
    long long sum = 0;
    for (int i = start; i < end; i++) sum += a[i];
    lock_guard<mutex> lock(sum_mutex);
    global_sum += sum;
}
int num_threads = 4;
vector<thread> threads;
int chunk_size = (n + num_threads - 1) / num_threads;
for (int t = 0; t < num_threads; t++) {
    int start = t * chunk_size;
    int end = min(start + chunk_size, n);
    threads.emplace_back(compute_sum, ref(a), start, end);
}
for (auto& th : threads) {
    th.join();
}
```

Mutual Exclusion in C++ with RAII

- Manually handling lock is not good

- You may forget to unlock
- Unlock is needed even when exception raised

- With RAII: `lock_guard<mutex>`
- Unlock itself once goes out of scope

```
mutex cnt_mutex;
int N = 100000;
for (int i = 0; i < N; i++) {
    // cnt_mutex.lock();
    lock_guard<mutex>
        lock(cnt_mutex);
    global_cnt++;
    // cnt_mutex.unlock();
}
```

Semaphores

- Extension of mutex (C++20)
- Allows at most N concurrent connections

- Example: Queue for parking space, wait to enter a store

Decrease count if success

Increase count

```
counting_semaphore<size_t> sem(5);
if (sem.try_acquire()) {
    // critical section
    sem.release();
}
```

Barriers

- Sometimes you want first complete all of step 1, then step 2
- Barrier: wait for threads to finish, then starts again

```
// spawn threads for step 1
for (int t = 0; t < num_threads; t++) {
    threads.emplace_back(step_1, t);
}
// barrier: wait for all threads to complete
for (auto& th : threads) {
    th.join();
}
// then continue step 2
for (int t = 0; t < num_threads; t++) {
    threads.emplace_back(step_2, t);
}
for (auto& th : threads) {
    th.join();
}
```

Atomics

- You can declare stuff as “atomic” (C++11)
- Operations are done as an entirety
- Won’t mess up even if multi-threaded

- Example: maintain counter in multi-threaded program

```
atomic<int> acnt;
void increment_counter() {
    for (int n = 10000; n; --n) {
        ++acnt;
    }
}
```

Atomics

```
Dump of assembler code for function _Z17increment_counterv:
   0x00000000000012e9 <+0>:     endbr64
   0x00000000000012ed <+4>:     push   rbp
   0x00000000000012ee <+5>:     mov    rbp,rsp
   0x00000000000012f1 <+8>:     sub    rsp,0x10
   0x00000000000012f5 <+12>:    mov    DWORD PTR [rbp-0x4],0x18
6a0
   0x00000000000012fc <+19>:    mov    DWORD PTR [rbp-0x8],0x0
   0x0000000000001303 <+26>:    jmp    0x131d <_Z17increment_co
unterv+52>
   0x0000000000001305 <+28>:    mov    esi,0x0
   0x000000000000130a <+33>:    lea    rax,[rip+0x4e43]
# 0x6154 <global_cnt>
   0x0000000000001311 <+40>:    mov    rdi,rax
   0x0000000000001314 <+43>:    call   0x1762 <_ZNSt13__atomic_
baseIiEppEi>
   0x0000000000001319 <+48>:    add    DWORD PTR [rbp-0x8],0x1
   0x000000000000131d <+52>:    mov    eax,DWORD PTR [rbp-0x8]
   0x0000000000001320 <+55>:    cmp    eax,DWORD PTR [rbp-0x4]
   0x0000000000001323 <+58>:    jl     0x1305 <_Z17increment_co
unterv+28>
   0x0000000000001325 <+60>:    nop
   0x0000000000001326 <+61>:    nop
   0x0000000000001327 <+62>:    leave
   0x0000000000001328 <+63>:    ret
End of assembler dump.
```

```
Dump of assembler code for function _ZNSt13__atomic_baseIiEppEi
:
   0x0000000000001762 <+0>:     endbr64
   0x0000000000001766 <+4>:     push   rbp
   0x0000000000001767 <+5>:     mov    rbp,rsp
   0x000000000000176a <+8>:     mov    QWORD PTR [rbp-0x18],rdi
   0x000000000000176e <+12>:    mov    DWORD PTR [rbp-0x1c],esi
   0x0000000000001771 <+15>:    mov    rax,QWORD PTR [rbp-0x18]
   0x0000000000001775 <+19>:    mov    QWORD PTR [rbp-0x8],rax
   0x0000000000001779 <+23>:    mov    DWORD PTR [rbp-0x10],0x1
   0x0000000000001780 <+30>:    mov    DWORD PTR [rbp-0xc],0x5
   0x0000000000001787 <+37>:    mov    edx,DWORD PTR [rbp-0x10]
   0x000000000000178a <+40>:    mov    rax,QWORD PTR [rbp-0x8]
   0x000000000000178e <+44>:    lock xadd DWORD PTR [rax],edx
   0x0000000000001792 <+48>:    mov    eax,edx
   0x0000000000001794 <+50>:    pop    rbp
   0x0000000000001795 <+51>:    ret
End of assembler dump.
```

Description ¶

ONE hardware instruction!

Exchanges the first operand (destination operand) with the second operand (source operand), then loads the sum of the two values into the destination operand. The destination operand can be a register or a memory location; the source operand is a register.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

Food for Thought

- Can you have `std::atomic<S>`?
- What about `std::atomic<T>`?

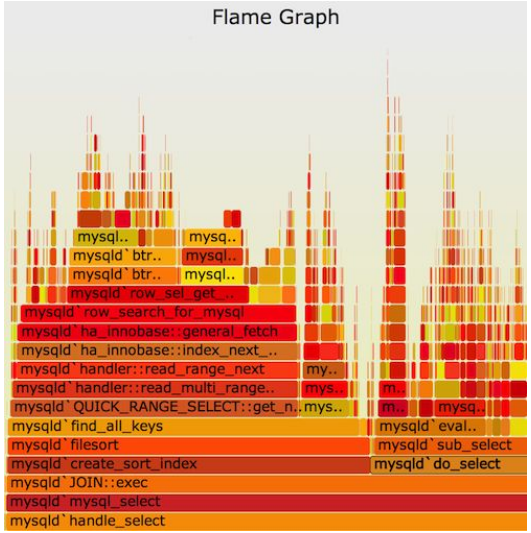
```
struct S {
  int l;
  int r;
};
```

```
struct T {
  int arr[1000];
};
```

Measuring Performance

Profiling with e.g. **flame graph**

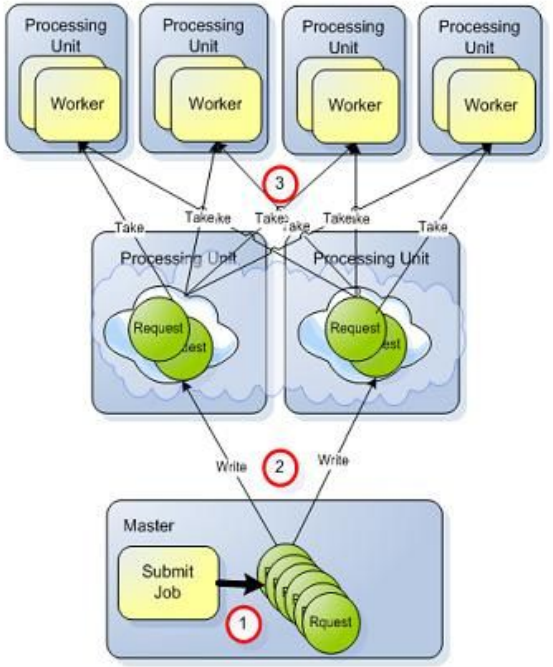
- Make the common case fast
- My code is slow because I allocate and deallocate too much memory.
- Object pooling



Design Principle - Master Worker

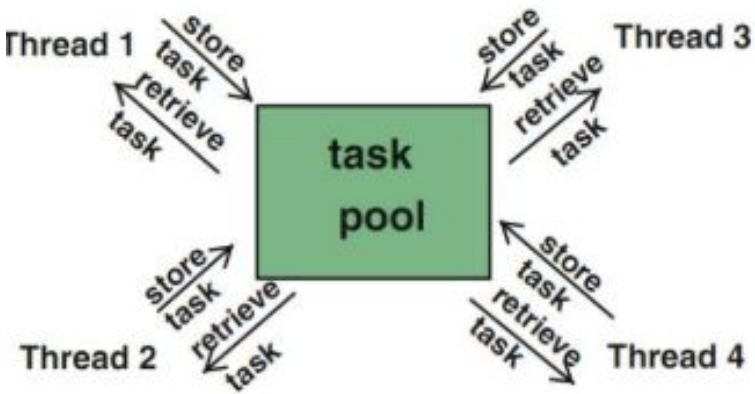
- Master receives all the jobs
- Master assigns the jobs to workers
- Workers further assign jobs to lower level workers

- Kind of like reality?



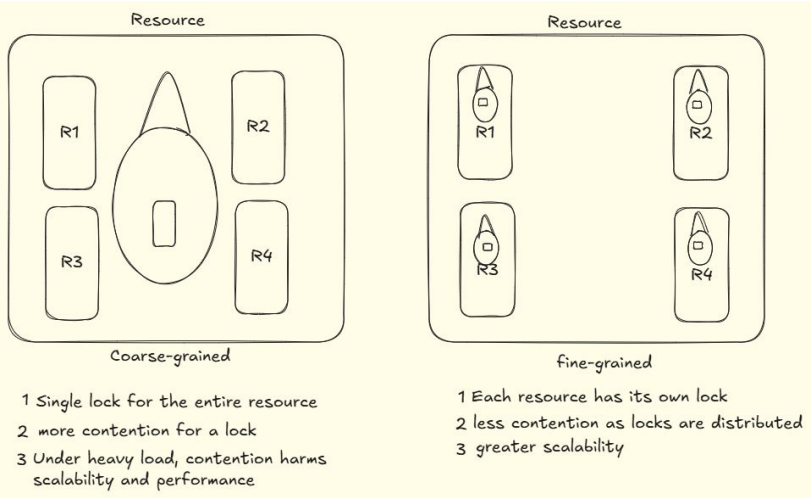
Design Principle - Task Pool

- Instead of pre-assigning jobs to thread, we instantiate a “pool of tasks”
- Once a thread done it original job, it take another job instead of quitting immediately
- So task distribution more even
- Also kind of like reality?



Design Principle - Fine-grained Locking

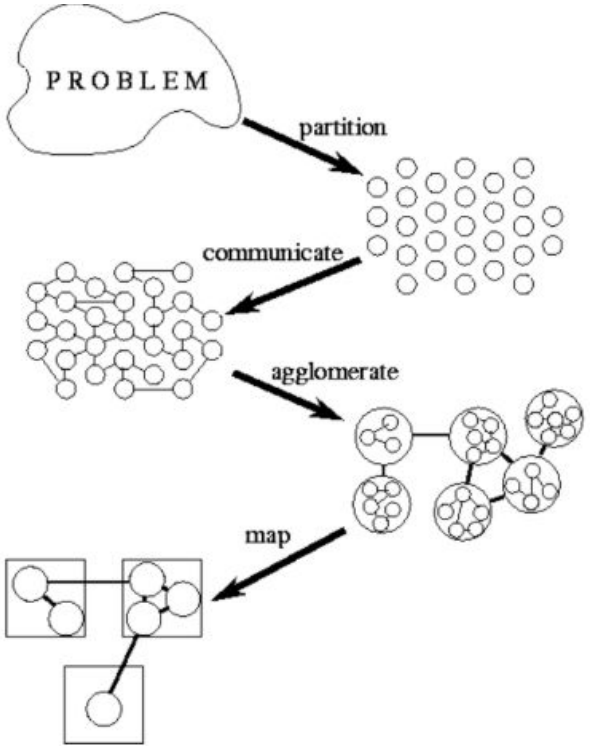
- Don’t lock the whole vector, just lock a part of vector / integer
- Less time needed to wait for unlock
- Example: we should not lock CUHK up for HKOI camp



How to design parallel algorithms?

1. **Partition** the problem into small tasks
2. Small tasks **communicate** with each other
3. **Agglomerate** tasks to groups to reduce communication cost
4. **Map** tasks to processors (cores)

Okay, enough theory... Let's move on to



Lab Exercises

Lab Exercises

Exercise 1 (~30 mins): M26E1 Matrix Multiplication

- Guided, mainly let you know how to play with parallelism

Exercise 2 (~1 hour): M26E2 Particle Simulation

- More up to you how to parallel the program!
- Rank by runtime :) Shortest runtime wins

- For both exercises, a sequential template code is provided

Lab Exercise 1. Matrix Multiplication

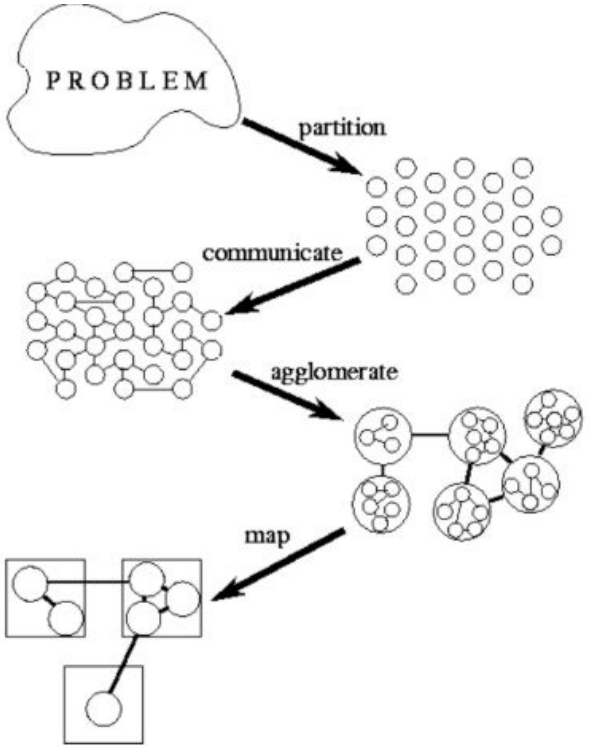
> Multiply matrices together.
> Be as fast as possible.

```
Input: int A[N][N], int B[N][N] Output: int C[N][N]
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        for (int k = 0; k < N; k++) {
            C[i][k] += A[i][j] * B[j][k];
        }
    }
}
```

Applying design principles

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        for (int k = 0; k < N; k++) {
            C[i][k] += A[i][j] * B[j][k];
        }
    }
}
```

Partition: `C[i][k] += A[i][j] * B[j][k];`
Communicate: Between same `C[i][k]`
Agglomerate: Merge those of same `i`/`j`/`k`?
Map: map those rows / columns to cores



Different loop orders

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        for (int k = 0; k < N; k++) {
            C[i][k] += A[i][j] * B[j][k];
        }
    }
}
```

```
for (int i = 0; i < N; i++) {
    for (int k = 0; k < N; k++) {
        for (int j = 0; j < N; j++) {
            C[i][k] += A[i][j] * B[j][k];
        }
    }
}
```

Is there a difference between different loop orders?

Different loop orders

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        for (int k = 0; k < N; k++) {
            C[i][k] += A[i][j] * B[j][k];
        }
    }
}
```

0.490s

```
for (int i = 0; i < N; i++) {
    for (int k = 0; k < N; k++) {
        for (int j = 0; j < N; j++) {
            C[i][k] += A[i][j] * B[j][k];
        }
    }
}
```

0.948s

- Continuous indices of B[j][k] are cached when read from memory
- Reading from cache is significantly faster than from memory
- Looping i,j,k will reduce the number of **cache misses**

Parallelising outer loops

1a. Alternate between threads for in the loop `i`

1b. Evenly split the `i` loop to uniform chunks, assign one to each thread

1c. Parallelise both `i` and `k` by giving each thread a range in each loop

- Easy to implement
- No overhead from syncing threads
- Best performance

Parallelising inner loop

The main bottleneck is that multiple threads may write to `C[i][k]`.

2a. Use **`std::lock_guard<std::mutex>`** to sync threads.

- Very poor in practice as high overhead from locking & unlocking

2b. Assign each thread an empty partial matrix to write to, then add all resulting matrices.

- Better, but still small overhead from combining results and creating empty matrices

Dynamic scheduling

```
std::atomic<int> next_cell{0};
while (true) {
    const int cell = next_cell.fetch_add(1, std::memory_order_relaxed);
    ...
}
```

3. Threads repeatedly claim the next `C[i][k]` to compute **atomically**
   - Balances work between threads well
   - Slow performance as high overhead from atomic operations

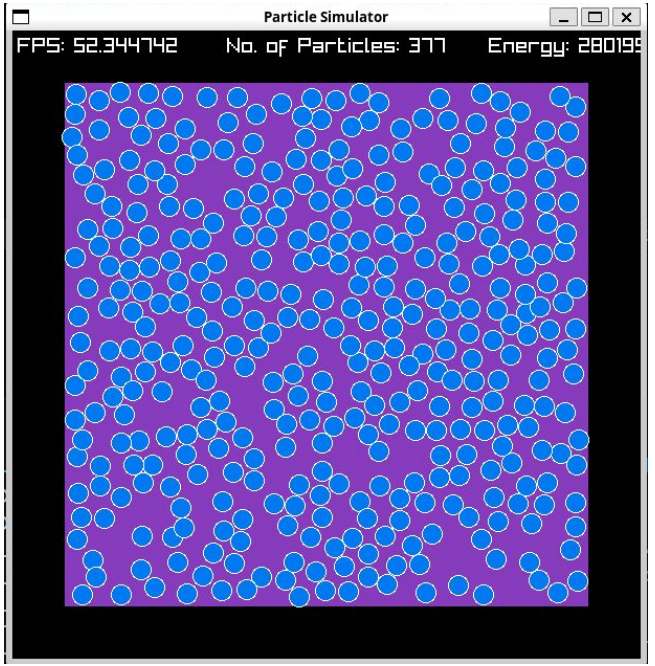
Benchmark Performances

1a. outerloop_one_thread_per_row	0.199s
1b. outerloop_parallel_row_chunks	0.235s
2a. innerloop_with_lock	0.353s
2b. innerloop_with_partial_matrix	0.346s
3. dynamic_scheduling	0.525s
Combination of 1c and 3 with cache optimisations	0.441s

Lab Exercise 2. Particle Simulation

> Simulate particle collisions.
> Retain high FPS with many particles.

```
// dt seconds passed since last frame
void update(double dt) { // called in while-loop
    move_particles();
    while (has_collision) {
        handle_particle_wall_collision();
        handle_particle_particle_collision();
    }
}
```



Particle-Wall Collision

Consider a particle at `pos` and velocity `vec`

After `dt` seconds:

- `pos_new = pos + vel * dt`

What if it collides with the wall?

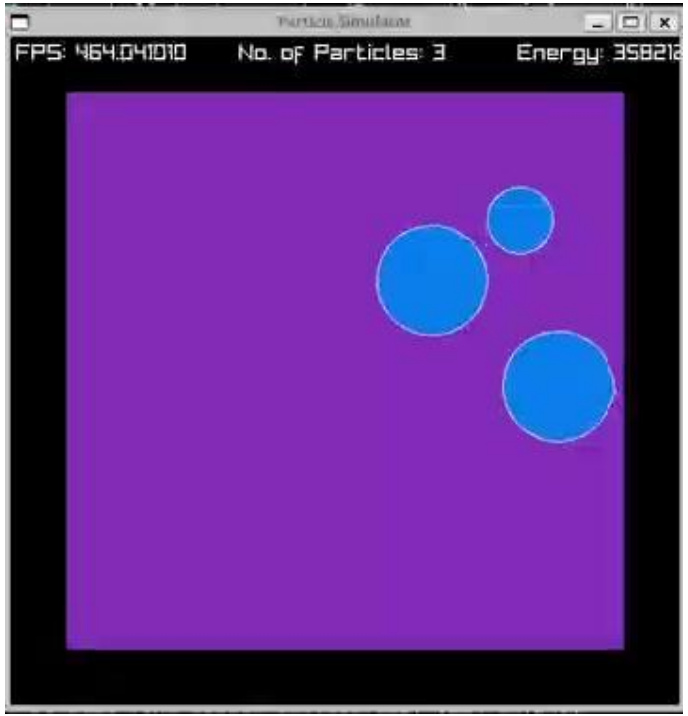
- Collide horizontally: `vel.x := -vel.y`
- Collide vertically: `vel.y := -vel.y`

Particle-Particle Collision

- According to physics, **energy** and **momentum** are always conserved
- Fact: this is used for check correctness

- So we can compute the velocity **after** the collision given the initial velocities

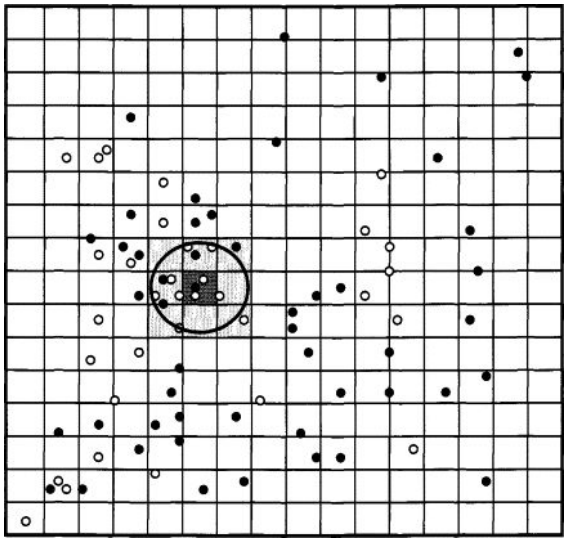
- The formulae are in the code so happily you don't need to deduce by yourself
- Sequential: $O(N^2)$, check for each pair



Recommendation: Spatial Partitioning

- **Observations**:
  - The particles should be distributed all across the grid.
  - Most pairs of particles are very far apart.

- Partition the grid into smaller cells!
- We only need to check collisions between adjacent / nearby cells.



Recommendation

- Implement spatial partitioning first. This gives you a speedup without applying any parallelism.
- Parallelize your code to utilize the 4 hardware threads available!
  - `static unsigned int thread::hardware_concurrency() noexcept;`
- Optimize! Profile your code and/or identify any inefficiencies.

References

Principle of Concurrent & Parallel Programming
https://assets.hkoi.org/training2023/parallel.pdf