Exam Assignments V02

chengchengguo 183090

1. What causes **false sharing**?

> "False sharing occurs when **threads on different processors modify variables that reside on the same cache line**. This invalidates the cache line and forces a memory update to maintain cache coherency."

when CPU wants to read a data from memory, the smallest unit of memory that can be transferred between the main memory and the cache is a **cache line** (typically 64 bytes).
when 2 CPU wants to read 2 different data in same cache line at same time, False sharing happens.

2. How do **mutual exclusion constructs** prevent **race conditions**?

> Mutual Exclusion： A form of synchronization in which **only one thread** at a time **can execute a block of code.**
>
> Usually a thread that reaches a mutual exclusion construct **waits until it gains exclusive access**.
>
> It is the requirement that **one thread of execution never enters a critical section while a concurrent thread of execution is already accessing critical section,** which refers to an interval of time during which a thread of execution accesses a shared resource, such as [Shared data objects, shared resources, shared memory].[1]

3. Explain the differences between **static** and **dynamic schedules in OpenMP.**

> The schedule clause affects **how loop iterations are mapped onto threads when using #pragma omp for**
>
> **static** [2]

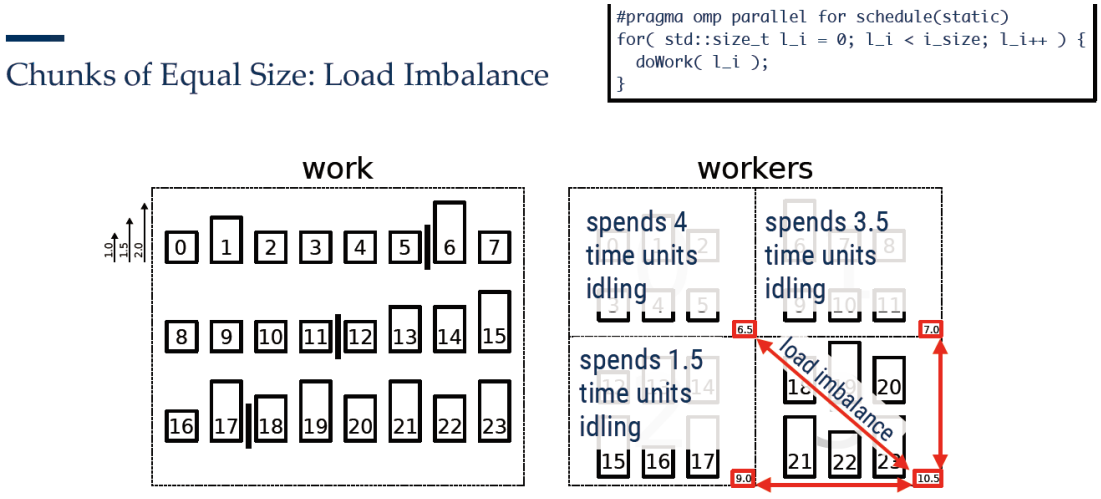
[1] Mutual exclusion - Wikipedia
[2] OpenMP-API-Specification-5-1.pdf

When kind is static, iterations are divided into chunks of size chunk_size, and the chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number. Each chunk contains chunk_size iterations, except for the chunk that contains the sequentially last iteration, which may have fewer iterations.
(When no chunk_size is specified, the iteration space is divided into chunks that are approximately equal in size, and at most one chunk is distributed to each thread. The size of the chunks is unspecified in this case.)

dynamic

When kind is dynamic, the iterations are distributed to threads in the team in chunks. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be distributed. Each chunk contains chunk_size iterations, except for the chunk that contains the sequentially last iteration, which may have fewer iterations.
When no chunk_size is specified, it defaults to 1.

pic: static may cause load imbalance [3]



If each iteration in the for loop needs the **same workload**, make the schedule **static**.
If the workload is **unbalanced between iterations**, make the schedule **dynamic**.

4. What can we do if we've **found a solution** while running a **parallel for loop** in OpenMP, but still have **many iterations left**?

Unfortunately, you **can't** easily just break out of a parallel for loop in OpenMP there is a work around:

[3] source: script from Parallel Computing (Alex Breuer)

continue_hack.cpp

```cpp
bool is_solution(int number) {  // test if number solves the problem
  for (volatile int i = 10000000; i--;) {}  // mock computation
  return number > 133 && number < 140;
}

int main() {
  constexpr int biggest_possible_number = 10000;
  // To avoid undefined code, it is necessary to use std::atomic for variables
  // where race conditions may happen.
  // https://databasearchitects.blogspot.com/2020/10/c-concurrency-model-on-x86-for-dummies.html
  atomic<bool> solution_found(false); // if true than we found the solution
  atomic<int> final_solution(INT32_MAX);
  const double start = omp_get_wtime();

#pragma omp parallel for schedule(dynamic)  // start parallel region
  for (int i = 0; i < biggest_possible_number; ++i) {
    if (solution_found)  // we found the solution, just continue iterating
      continue;
    if (is_solution(i)) {     // find some solution, not necessary the smallest
      solution_found = true;
      final_solution = i;
    }
  }  // end parallel region
  // check if we've found a solution at all is omitted, you can add the check
  cout << "The solution is: " << final_solution << endl;
  cout << omp_get_wtime() - start << " seconds" << endl;
}
```

in this example it uses **is_solution**(bool type) to check if we found the answer, define a atomic (because we want to avoid race condition, so we won't give the result of different thread to the parameter, that keeps the result , at the same time.) variable **solution_found**.

we check in every thread if it is the answer, if it is, we jump out of the **#pragma omp parallel for** loop