Algorithm Engineering
Exam Assignments

Sheela Orgler

Friedrich Schiller Universität, Jena

1 Assignment

1.1 Describe how parallelism differs from concurrency

Parallelism is a subset of concurrency. If a system is parallel it is also concurrent, but not the other way around. A system is considered parallel if two or more tasks are executed simultaneously.
Concurrency only means support for two or more actions at the same time. Parallelism refers to execution of more than one action at the same time.

1.2 What is fork-join parallelism?

Fork-join parallelism refers to a concept where we have a master thread that divides into a team of threads. With this concept we only have parallel regions where threads are executed simultaneously. In between these parallel regions the execution is sequential following the master thread.
We can imagine execution as a line that is followed from starting to end point. With fork-join parallelism this line is our master thread. Following our master thread at some point the thread can divide into two or more threads. These threads are joined after some time to our master thread. In these sections where we have two or more threads execution will be parallel and after these sections sequential.

1.3 Chapter 1 – Computer Systems: A Programmer's Perspective

Discuss one thing - Caches As discussed in Chapter 1 of the book caches are helpful to deal with the processor-memory gap.

The processor can read data from the register file, which is within the CPU almost 100 times faster than from memory. The gap between processor and memory is continuously growing.

Cache memory is used to bridge the gap. It is used to temporarily store information that will likely be used in the future. There exist several levels of caches - L1, L2, L3 - which go from smallest to largest.

A cache reduces the acess time to data in memory. Frequently used data and instructions are kept in the cache. Access to data kept there is faster.

Modern desktops, servers and industrial CPUs have at least three independent caches: instruction cache, data cache and the TLB. The instruction cache is used to speed up executable instruction fetches. The data cache speeds up data fetch and store. It is organized into more cache levels:

– L1 cache - Primary cache is fast but small and usually embedded in the processor chip (CPU)
– L2 cache - Secondary cache is larger and can be embedded in the CPU or on a seperate chip or coprocessor containing a high-speed alternative system bus connectin cache and CPU.
– L3 cache - specialized memory to improve L1 and L2. L3 is usually double the speed of DRAM

The small caches L1 are backed up by larger and slower caches L2, L3 to address the tradeoff between cache latency and hit rate. The faster cache is generally checked firt. If that cache misses, the next chache is checked and so on, before accessing main memory.

For example the ARM-base Apple M1 CPU has 8 cores: 4 high-performance and four high-efficiency cores. The four high-performance cores have a 192 KiB L1 cache for each of the cores and the four high-efficiency cores only have 128 KiB.

The Translation lookaside buffer is used to speed up the translation from virtual to physical addresses. It is part of the memory management unit and not directly related to CPU caches.

1.4 Paper – There's plenty of room at the Top: What will drive computer performance after Moore's law?

Explain figure "Performance gains after Moore's law ends" The figure illustrates the "Top" and "Bottom" referring to computer performance gains. The "Bottom" refers to the miniaturization of computer components seen in the last decades. Due to physical limits the opportunities for gains at the bottom will slowly come to an end. Nevertheless, there are still opportunities for growth at the "Top". The "Top" shows the three aspects where growth can be expected: software, algorithms, hardware architecture. These aspects are divided into technology, opportunity and examples all looking into the growth opportunities in the specific area.
Growth opportunities at the "Top":

– making software more efficient by performance engineering
– minimizing the time it takes to run and not the development time
– an increasing number of processor cores running parallel
– reengineer modularity to obtain performance gains

2 Assignment

2.1 What causes false sharing?

False sharing occurs, when threads on different processors modify variables on the same cache line. A cache line is the smallest unit of memory. Its length

depends on the underlying architecture and is typically 64 and in more recent architectures 128 bytes long.
False sharing happens when for example two threads on different processors modify variables on the same cache line. The first thread modifies one variable at the beginning of the cache line. The second thread tries to modify a variable at the end of the cache line. Due to the alteration of the first thread, the cache line is invalidated for the second thread and has to be reloaded.
False sharing only happens when variables are changed and not when they are only read. It may lead to a significant performance decrease.

2.2 How do mutual exclusion constructs prevent race conditions?

Race conditions occur when two threads modify the same data. Mutual exclusions give one thread exclusive access to the data. After execution exclusive access is returned and the next thread can execute.

2.3 Explain the differences betweent static and dynamic schedules in OpenMP.

Static and dynamic schedules differ in how the work $iterationsofthe for loop)$ is spread across the threads. Static means that it is decided at the beginning and dynamic means that it is decided at runtime. Each thread will work on a chunk of values and then take the next chunk that hasn't been worked on by any thread. Static schedules perform better for balanced workloads and dynamic schedules for unbalanced workloads, in case the workload varies between different iterations of the for loop. The chunk size can be specifies; for static schedules it is one per default.

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

If we've found a solution within a parallel for loop, we can use continue. We don't continue calculating, but quickly iterate through the loop to increase performance.

2.5 Explain in your own words how std::atomic::compare_exchange_weak work

strd::atomic::compare_exchange_weak succeeds only if the value of the variable to be updated equals the first provided argument. For example if for final_solution.compare_exchange_weak(previous, i) final_solution equals to previous. Previous is updated with i, if it fails.

3 Assignment

3.1 How does ordered clause in OpenMP work in conjunction with a parallel loop?

An ordered clause is used within a parallel region. It opens a region where the execution is sequential, in order to prevent race conditions.

3.2 What is the collapse clause in OpenMP good for?

The collapse clause is used to parallelize for loops. It allows to specify how many loops are collapsed into one. The collapse clause is useful for balancing work and parallelize multiple loops to boost performance. When using the collapse clause a single loop is formed. The length of the new loop is equal to the multiplication of the length of each loop that has to be collapsed into one.

3.3 Explain how reductions work internally in OpenMP

A reduction is an operation form OpenMP that is used in parallel computing. It reduces the calues into a single result. The syntax of the operation is: reduction(op: list). op refers to the operation, which can be +, -, *, and so on. list refers to a list of variables that is reduced. For example: A reduction can be used within a parallel for loop: #**pragma** omp parallel **for** reduction(+ : sum) to compute the sum of the values 1, 2, 3. Every thread within the parallel region gets a local sum variable, where the sum is computed. The result will be written to the global sum variable declared outside of the parallel region.

3.4 What is the purpose of a barrier in parallel computing?

A barrier can be set within a parallel region. Setting a barrier means that the threads will execute the code until that specific barrier. There they wait until all threads have reached the barrier and only then continue with execution. Barriers are used for synchronization in parallel regions, where they enforce a specific execution order.
Within OpenMP there are also implicit barriers. The most general barrier is the omp parallel region itself. The region makes sure that execution is parallel within that region. After the code from that region is executed in parallel execution will continue sequentially.

3.5 Explain the difference between the library routines: omp_get_num_threads(), omp_get_num_procs() and omp_get_max_threads().

- omp_get_num_threads() - The routine is called inside a parallel region. It is used to get the number of threads used within a parallel region.

– omp_get_num_procs() - The routine is used to get the number of logical cored. It can be used to define the number of threads reasonable to use in a parallel region. Per default the number of threads is equal to the number of logical cores.
– omp_get_max_threads() - The routine is called outside a parallel region and returns the number of threads supported in a parallel region. When the value is changed with omp_set_num_threads() within a parallel region, the method returns the set value.

3.6 Clarify how the storage attributes private and firstprivate differ from each other.

Storage attributes are used to specify the usage of variables within a parallel region. It is used for variables declared outside a parallel region. There are three different storage attributes: shared, firstprivate and private.

Private attributes make sure that each thread gets an uninitialized copy of the variable. Firstprivate attributes make sure that eacht thrad gets an identical initialized copy of the variable. The variable is a local variable for each thread. When using private or firstprivate storage attributes the value of the global variable remains unchanged.

3.7 Write in pseudo code how the computation of pi can be parallelized with simple threads.

```
initialize num_steps to 100000000
initialize width to 1.0
initialize sum to 0.0

do in parallel:
  initialize sum_local to 0.0
  for i in {0, num_steps} do
    set x to ((i + 0.5) * width)
        add sum_local to (1.0 / (1.0 + x * x))

        do sequentially:
          add sum sum_local

initialize pi to sum * 4 * width
return pi
```

4 Assignment

4.1 Explain how divide and conquer algorithms can be parallelized with tasks in OpenMP.

Divide and conquer algorithms recursively split down a problem until the pieces can be directly solved. Such algorithms can be paralleized in OpenMP using the constructs taks and taskwait for the recursive call. OpenMP tasks are independent work blocks. A queueing system dynamically handles the assignment of threads to the tasks. Each threads picks up a task from the queue and executes it. This goes on until the queue is empty. Tasks are defined within a parallel region and within a single construct, to make sure that one thread creates the tasks. Taskwait is a construct that waits until all tasks are executed.

4.2 Describe some ways to speed up merge sort.

- Tasks - With task and taskwait
- Avoid copying - to prevent that additional memory is created recursively and make sure that initially memory is instantiated and reused.
- if Clause - task will only be created, if a certain condition is true
- final Clause - tasks won't be created anymore, if a condition is true

4.3 What is the idea behind multithreaded merging?

The goal is to sort and merge two arrays using a divide and conquer algorithm. To do that we look at the median of the array with more elements. Then we look at the other array and compare the elements smaller than the median and merge these with the respective elements from the other array. That way merge is executed in parallel.

4.4 Read What every systems programmer should know about concurrency. https://assets.bitbashing.io/papers/concurrency-primer.pdf Discuss two things you find particularly interesting.

The order of written code can be changed as compiler try to optimize it and rewrite code to run faster on the targeted hardware. RAM hasn't speeded up the way CPU processors have. This creates a widening gap between the instruction fetch and the time needed to retrieve data from memory. To prevent reordering we can use atomic types in C or C++.

Atomicity means that something can't be divided into smaller pieces. Threads need to use atomic reads and writes to share data. If they are not atomic they are so-called torn reads and writes. To ensure atomicity one need to make sure that variables used for thread synchronization aren't larger than the CPU word size.

References

1. Armv8-A Reference Manual pdf, pages 30-70, pages 1708-1808 https://developer.arm.com/documentation/ddi0487/ga.