UE18CS342
Heterogeneous Parallelism
Unit 3

Aronya Baksy

March 2021

1 OpenMP

- Open Specification for Multi Processing, uses threads and shared memory to allow parallel execution of tasks.
- It is implemented for C/C++/Fortran. The main components of OpenMP are
  - Compiler Directives
  - Runtime library routines
  - Environment variables
- Features of OpenMP:
  - Communication via shared memory: variables may be unique to a thread or shared between all threads
  - Explicit parallelism (representation of concurrent computations by means of primitives in the form of special-purpose directives or function calls.)
  - Fork-join model (start with one thread, spawn multiple threads for the parallel part, then join all of them back to the main thread)
- Unintended sharing of data can lead to race conditions (when the program's outcome changes as the threads are scheduled differently)
- Synchronization primitives are used to control race conditions. But improper use of such primitives leads to deadlocks.

```c
#include<stdio.h>
#include<omp.h>
int main()
{
    #pragma omp parallel
    {
        //Print the thread ID of the current thread
        //Number of threads = OMP_NUM_THREADS
        printf("Hello from thread number %d\n", omp_get_thread_num());
    }
}
```

Figure 1: OpenMP code example

1.1 OMP Runtime Library functions

- The `omp_get_thread_num()` function returns the thread ID of the current thread (returns an int value between 0 and numthreads-1)
- The `omp_get_num_threads()` function returns the total number of threads
- The `omp_set_num_threads(n)` function sets the total number of threads for all upcoming parallel regions.
- The `omp_get_num_procs()` function returns the total number of processors

1.2 OMP Directives

- The directive `#pragma omp parallel [clause [clause ]...]` and the flower brackets after it denote the parallel region of the program.
- `clause` is either `private(varlist)`, `shared(varlist)` or `default(type)`
- The default clause allows to specify the default type of variables inside the parallel region.
- The directive `#pragma omp for` distributes the next for loop it encounters across the threads created. If it is placed aobve a nested loop then it applies only to the first outer loop and not the inner loops after it (the inner loops still run in serial on each thread from start to end)

1.3 OMP Work Sharing Constructs

C/C++ Syntax	**Functionality**
`#pragma omp for`	Distribute iterations over the threads
`#pragma omp sections`	Distribute independent work units over threads
`#pragma omp section`	One work unit that is executed on one thread
`#pragma omp single`	Only one thread executes a code block. The other threads wait at a **barrier** until the thread executing the single code block has completed

1.4 OMP Shared and Private Variables

- In the `#pragma omp parallel` the clause fields allow one to specify a list of shared and private variables.
- By default, all variables are **shared** between the threads. If `default(none)` is specified then each variable has to be declared explicitly inside parallel region.
- Generally read only variables are shared (they can not accidently be over written) and main arrays are shared.
- A variable that is **private** can only be read or written by its own thread.
- When a variable is declared as private, a new object of the same type is declared once for each thread and all references to the original object are replaced with references to the new object.
- Loop indices and loop temporaries are generally made private.
- **Threadprivate** variables are variables that are private to one thread only, but persist even after the completion of the parallel region. This is possible as they are declared on the heap instead of the stack.
- The value of a threadprivate variable after the parallel region is the value it had in the initial master thread.

1.5 OMP Storage Associations

- Private variables inside a parallel region have no association with the same variables outside the parallel region (private vars are undefined on entry and exit of parallel region)
- The **firstprivate** clause specifies that each thread should have its own instance of a variable, and that the variable should be initialized with the value of the variable, because it exists before the parallel construct.
- The **lastprivate** clause specifies that the enclosing context's version of the variable is set equal to the private version of whichever thread executes the final iteration (for-loop construct) or last section.
- **Disadvantage**: performance penalty in using lastprivate becuase openMP needs to keep track of the value and which thread executed the last iteration (easy for static workloads but costly for dynamic workloads)

1.6 OMP Scheduling

- The `schedule` clause allows one to specify how loop iterations should be scheduled between threads. The default scheduling behaviour (in the absence of a schedule clause) is stored in the `def-sched-var` variable
- The general syntax of the schedule clause is `#pragma omp <dir> schedule(type, chunk)`.

1.6.1 Scheduling Types

- **Static**: The loop iteration space is divided into chunks of size `chunksize` and assigned in a round-robin manner to each thread (in order of their thread ID). When no chunk size is specified the chunk size is automatically calculated (most evenly across threads)
- **Dynamic**: Chunks of iterations are assigned to threads as they are requested by the thread. Each thread executes a chunk and requests another until no more chunks are available.
- Dynamic scheduling has higher overheads due to runtime distribution but it is appropriate if the thread workloads are not balanced.
- **Guided**: Similar to dynamic scheduling in that chunks are assigned to threads on request until all chunks are exhausted. The chunk size is calculated on the fly
- The chunk size is proportional to the number of unassigned iterations divided by the number of the threads. Therefore the size of the chunks decreases exponentially as work proceeds
- Minimum size of a chunk is chunksize (specified) except for the last chunk which can have size less than chunksize.
- **Runtime**: The scheduling decision is deferred till runtime. The `OMP_SCHEDULE` environment variable and the `omp_set_schedule()` function are used to change this.

1.7 Limitations of OMP Threads

- Even though more number of threads are available, the sections are limited to the declarations.
- Assuming two sections that execute `funcA()` and `funcB()`:
  - If only one thread is available, both calls are executed in sequential order.
  - Depending on the type of work performed in the various code blocks and the number of threads used, this might lead to a Load-Balancing problem.
  - This occurs when threads have different loads and thus take different amounts of time to complete.
  - A result of load imbalance is that some threads may wait a long time at the next barrier in the program, which means that the h/w resources are not being utilized efficiently

2 Parallel Programming

2.1 Loop Optimizations

- **Loop Interchange** is applicable in case of nested loops. When a 2D array is being accessed in column major order, the cache utilization takes a hit (assuming that data is loaded from memory in row-major order). In this case swapping the inner and outer loops increases performance.
- Loop overheads come from pointer arithmetic that has to be done in case of array indexing accesses.
- Overheads are high when each loop has small number of operations. **Loop unrolling** puts more operations in each iteration and reduces number of iterations, therefore overheads are reduced.
- **Loop Fusion** (combining separate loops into one when feasible) increases cache utilization and hence reduces memory access latencies. Apply when same array is manipulated in separate loops.
- **Loop Fission**, i.e. splitting one loop into separate loops in an effort to increase cache utilization, can also be applied.
- **Loop tiling** partitions a loop's iteration space into smaller chunks or blocks, so as to help ensure data used in a loop stays in the cache until it is reused.
- Loop tiling leads to partitioning of a large array into smaller blocks, thus fitting the accessed array elements into cache size and increasing cache utilization.

2.2 Memory Models

2.2.1 Shared Memory

- Threads share a common address space that is concurrently read from/written to
- Access control for concurrent operations is maintained via primitives such as locks/mutexes/semaphores
- Advantage: there is no ownership of memory by any one task hence no need to explicitly specify the flow of data between the tasks
- Disadvantage: Difficult to manage data locality (performance improvement by keeping data local to the same processor, reduces memory access, cache access, bus traffic)

2.2.2 Message Passing Model

- Tasks residing on either the same machine or arbitrary number of physical machines, each using their own local memory during computation
- Data exchange between tasks is done by sending and receiving messages
- Advantage: easier to implement than shared memory, and tolerant to high latencies in communication
- Disadvantage: Slower communication than the shared memory model because of connection setup overhead

2.2.3 Data Parallel Model

- The working data is organized in a single common data structure. Each task works in parallel on different partitions of the data structure
- On distributed memory systems, the data structure is split into chunks with each chunk residing on the local memory of each task.
- On shared memory systems all tasks have access to the data via the global memory.

2.3 Memory Consistency Models

- The Memory Consistency model of a shared memory multiprocessor provides a formal specification of how the memory system will appear to the programmer, eliminating the gap between the expected behaviour and the actual system behaviour
- Superscalar processors can reorder memory accesses, and executes 2 memory accesses in sequence as long as they don't modify the same data.
- A processor may reorder memory accesses e.g. due to lockup-free caches, multiple issue, or write buffer bypassing.

2.3.1 Lockup-Free Caches

- In such a cache, if one cache miss occurs then the cache can still continue to serve other cache requests instead of locking the memory bus waiting for the fresh data to arrive.
- This allows for reordering of memory accesses. Superscalar processors make use of this to reorder, thus in the event of a cache miss the execution continues until the data from the cache miss is actually needed again.
- Data prefetching caches load memory predictively before it is to be used. Such behaviour requires a lockup-free cache (aka a non-blocking cache)

2.3.2 Write Buffer Bypassing

- First level write buffer exists between L1 and L2 cache (allows L2 to service reads in the meantime). Second level write buffer exists between L2 and the memory bus interface
- By letting read misses bypass writes in the buffer to other addresses, the reads can be serviced faster. This involves reordering memory accesses.
- A superscalar processor has queues in the load-store functional unit where accesses also can be reordered.

2.4 Sequential Consistency

- A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and this order is the same as the order specified in the program.
- Basically, a write to a variable by any processor needs to be seen instantaneously by all processors.
- Intuitively reasonable reordering of memory operations in a uniprocessor may violate sequential consistency model
- Modern microprocessors reorder operations all the time to obtain performance (write buffers, overlapped writes,non-blocking reads...).

2.5 Relaxed Consistency

- All memory operations are classed as either synchronization or data operations.
- A sync operation is like a fence. All data operations before a sync operation must be completed before the sync is executed. Data operations before a sync can be reordered
- Sync operations always in program order, by disallowing reordering of operations around them.
- The Flush directive in OMP provides the strength to synchronization operation `#pragma omp flush (list)`. It tells the OpenMP compiler to generate code to make the thread's private view on the shared memory consistent again.
- The flush operation implicitly takes pace around:
  - Barriers

  - Entry/Exit from Parallel, Parallel work-sharing, critical regions
  - Lock functions
  - Entry to and Exit from atomic operations

2.6 Release Consistency

- A further weakening of the weak ordering that is implemented in relaxed consistency.
- All sync operations have 2 parts:
  - **Acquire**: Acquire must complete before all following memory operations
  - **Release**: All memory access operations before release must complete, accesses after release in program order need not wait for release.
- Assume that two processors P1 and P2 hold a lock L.
- In the **early release**, P1 after Release(L) broadcasts value of L to all processors for coherence operation. Even processors who do not have x will get the message
- In the **lazy release**, No broadcast when P1 release the lock, but when P2 acquires the lock, the system has to make sure all coherence before acquire of lock has to be completed. Reduces messages hence less bandwidth consumed.