CMSC416: Introduction to Parallel Computing

Topic: OpenMP
Date: February 29, 2024

Previous Slides Review

OpenMP is a **fork-join model**, in which threads are created, run in parallel, and then rejoined to the master thread.
OpenMP **pragmas** are structured as `#pragma omp construct [clause [clause]...]`, where "omp" is an OpenMP keyword, construct being "parallel" or "parallel for", and clauses following are optional.
The **"parallel for"** construct parallelizes the for-loop immediately following the pragma line. It takes the loop iterations and assigns them to different threads, which run in parallel.

Number of Threads

There are two ways of setting the number of threads:

1. The number of threads can be set on the **command line** before the executable is launched using the environment variable `export OMP_NUM_THREADS=X`, where X is your specified number of threads. This is done for the entire program execution.
2. The number of threads can also be set differently for different parallel regions with the **library routine** `void omp_set_num_threads (int num_threads)`. It can be called before different parallel regions to change the number of threads for the subsequent region.

The function `int omp_get_num_procs(void)` may be used to know how many physical cores there are on the node. It uses hardware identification libraries to determine how many available cores there are.

Data Sharing Defaults

We are only looking at parallelizing **for-loops** in OpenMP in this course.
Most variables in OpenMP are **shared by default**, including global variables. This is different from MPI, where each process gets its own virtual address space and processes cannot directly access other processes' memory. In a threaded program by OpenMP, all threads can access most items.
One of the only exceptions is **loop-index variables**, which are private by default. Every thread has its own private copy of a variable (for example, the variable i in the for-loop). If loop-index variables were shared, the threads would modify the variable in a random order and keep writing to it over each other. OpenMP aims to assign a different number of iterations to different threads, and to do this, it ensures that the loop-index variable is private to each thread.
Another exception is **stack variables in function calls from parallel regions**. These are private to each thread, since similar to loop-index variables, if they were shared among threads, they would be overwritten by threads.
Thread-private is a term often used in OpenMP to describe these variables as they are private to threads rather than shared. Loop-index variables and stack variables in function calls are both thread-private.

saxpy (single precision a*x+y) example

The **saxpy** example is a 32-bit computation in which two arrays are added together where one of the arrays is multiplied by a constant. In the for-loop shown on the slide for saxpy, a value a*x[i]+y[i] is created at each index i, and assigned to a value z[i].
We do not expect race conditions here when different parts of the loop are assigned to different threads, because the `#pragma omp parallel for` line was added before the for-loop, and one array is used for writing and the other two are used for reading all at the same index.

Clauses

Clauses are additional elements you can add after a pragma directive to override default specifications.

1. **Private Clause** (`private(list)`) - Each thread has its own private copy of the variables of the list. Since most variables are shared by default, this allows you to specify to make variables private to the threads. These private variables are fully uninitialized when a thread starts, so you'll have to give it values in the first loop of the for loop iteration. The value of a private variable is unavailable to the master thread after the parallel region has been executed.
2. **Default Clause** (`default(shared | none)`) - The OpenMP runtime will determine the data-sharing attributes for the variables, where they would have been implicitly determined otherwise.

Anything wrong with this example?

1. First Example
    a. In this case, a value of 5 was assigned to a variable val by the master thread and then that variable was made private with the private clause. This gives each thread a new, uninitialized variable of val, so it would not be right to assign the value beforehand. That value of 5 assigned before will not be available inside the for loop.
2. Second Example
    a. Here, a variable val was made private and was used in the for-loop and then accessed after. When the parallel region is exited, this variable is out of scope, so the value of it will not be available to the master thread outside the loop.

Clauses (cont.)

OpenMP has other clauses that can let you access these private values.

3. **Firstprivate Clause** (`firstprivate(list)`): This clause initializes each thread's private copy to the value of the master thread's copy. Taking the first example above, where the variable val is set to 5 by the master thread before the for loop, adding the firstprivate clause before the for-loop will initialize the value of val to 5 for all threads.
4. **Lastprivate Clause** (`lastprivate(list)`): Using this clause makes the value inside a region available to outside, the master thread, through writing the value of the variable

from the thread as it was in the last iteration of the for-loop to the master's copy of this variable.
    a. For example, if n was 100, and Thread 0 executed iterations 0-24, Thread 1 executed iterations 25-49, Thread 3 executed iterations 50-74, and Thread 2 executed iterations 75-99, the value will be taken from Thread 2, since it executes the last iteration of 99.
    b. We only take the value from the thread that executes the last iteration (regardless of thread ID). The last iteration is determined by sequential order (in this case, index 99 since it is the highest number).
5. **Reduction Clause** (`reduction(operator: list)`): This clause is used when there is a need for an operation to be performed over the loop.
    a. In the first case, val is a shared variable, where inside the for-loop, val is assigned to the sum of i and val. Here, since every thread shares val, there is a race condition here, since every thread is trying to read to and write from val.
    b. Normally, you could make val private, but then you'd have to sum all of the values after the for loop.
    c. The reduction clause fixes this issue by taking values across private thread copies of each variable and reducing them to one global variable at the end of the for-loop by performing the specified operation. Two arguments must be provided in the reduction clause:
        i. A list of variables that you want to do the reduction over.
        ii. An operator that you want to perform on all of the variables (in this example, sum).
    d. For operations, depending on which operator it is, it decides initial values for omp_priv. It is usually 0, but in cases like multiplication where 0 would ruin the operation, it initializes it to 1.

Loop Scheduling

OpenMP uses **loop scheduling** to assign threads to different parts/iterations of the loop.
OpenMP's default schedule attempts to balance the iterations among threads.
Note: If different iterations do different amounts of work, it will not be load balanced using the default schedule, but if each iteration does around equal amounts of work, this default schedule will balance out the work.

User-Specified Loop Scheduling

6. **Schedule Clause** (`schedule (type[, chunk])`) - This clause allows for users to specify loop scheduling. There are 4 popular types of this:
    1. Static: The number of iterations are divided as evenly as possible. (# iterations / # threads)
        a. You can use the argument **chunk** to specify a smaller number of iterations for each thread so that they can come back and get assigned more iterations. A round-robin scheme is an example of this where chunk is 1.
    2. Dynamic: A chunk size block is assigned to each thread.

a. Whichever thread is done first will come back and ask for the next chunk of work from the OpenMP runtime.
    i. This is as opposed to the static case, where the threads come back in order of which they are assigned.
b. This promotes load balance by ensuring that threads with less work can finish and retrieve more work. It has a default chunk size of 1.