CMSC416: Introduction to Parallel Computing

Topic: OpenMP and Parallel Algorithms
Date: March 05, 2024

Loop Scheduling

Loop scheduling is the assignment of loop iterations to different worker threads. The user can specify the type using the schedule clause:

```
schedule (type[, chunk])
```

The type of the schedule can be static, dynamic, guided, or runtime. The default schedule tries to balance iterations among threads. In a static schedule, iterations are divided as evenly as possible before the program runs. In a dynamic schedule, OpenMP assigns a chunk size (default size 1) block to each thread, and once a thread finishes a block, it retrieves the next block from an internal work queue. This allows balancing work across threads even though the iterations were not evenly divided beforehand. A guided schedule is similar to a dynamic schedule, but it starts with a large chunk size and gradually reduces it to handle load imbalance between iterations. Lastly, an auto schedule is where scheduling is delegated to the compiler.

When the user sets the schedule at runtime using the OMP_SCHEDULE environment variable, the schedule can be changed later on without re-compiling the program. However, if it is set in the program, it is hard-coded there and the program must be re-compiled if the user wants to change the schedule. (This is different from setting the OMP_NUM_THREADS environment variable, where once it is set, the entire program uses that value whereas if it is set in the program itself, the number of threads can be changed for different sections of code.)

Example: Calculate the value of pi

Consider the code block below:

```
int main(int argc, char *argv[])
{
      ...
      n = 10000;

      h   = 1.0 / (double) n;
      sum = 0.0;

      for (i = 1; i <= n; i += 1) {
            x = h * ((double)i - 0.5);
            sum += (4.0 / (1.0 + x * x));
      }
      pi = h * sum;
      ...
```

```
}
```

Using OpenMP, we want to assign different iterations to different threads. Let's first consider which variables might get overwritten by different threads. The variables `sum` and `x` will be overwritten as they are by default shared, so we need to make them private. Thus, we add the following pragma before the for-loop:

```
#pragma omp parallel for firstprivate(h) private(x) reduction(+: sum)
```

We want every thread to have a private copy of `sum`, and at the end of the loop we want to combine all the sums and make the reduced value visible to the master thread. Note that the `firstprivate(h)` included is not necessary as even though the variable `h` is shared by default, it is read-only in this program so we do not need to make it private.

Parallel Region

Other than for-loops we can also instruct all threads to execute a single statement or structured block of code. However, this requires more manual work for the user. In the case of parallel for, the OpenMP runtime knows the number of iterations and divides those across threads. In the case of parallel regions, there is no way for the compiler to auto-parallelize. The user needs to manually decide which thread(s) will execute what part of that structured block (Note that if the statement is just a single printf, there is no need to set anything since every thread will just run the print statement, but if the code block is more complex, the user needs to manually parallelize). Similar to parallel for, the number of threads can be specified by the user. Note that OpenMP is more commonly used for programs with for loops as the primary use case.

Synchronization

We need synchronization because concurrent access to shared data may result in inconsistencies. We avoid this using mutual exclusion - in parallel regions we may need to use other directives inside to prevent overwriting. Mutual exclusion means that only 1 thread writes to a certain variable at a time before other threads can read its value. We discuss 2 directives:

The critical directive specifies that the following code is only to be executed by 1 thread at a time (note that this should only be used for small portions of code as it essentially makes the code sequential). It is written:

```
#pragma omp critical [(name)]
      structured block
```

The atomic directive specifies that a memory location should be updated by 1 thread at a time before other threads can read. It is written:

```
#pragma omp atomic
      expression
```

GPGPUs: General Purpose Graphical Processing Units

The difference between GPUs and CPUs lies in that GPUs have a larger number of low power (slower) cores, allowing us to parallelize computation over a larger number of cores. There are maybe 10,000 cores on a modern GPU. GPUs are accelerators and typically attached to a CPU as a helper. Thus, we often need to copy data from CPU memory to GPU in order to perform computations and then copy the data back into CPU memory.

We can direct OpenMP to use the GPU for a for loop using the following pragma:

```
#pragma omp target teams distribute parallel for
```

The target keyword means to run on an accelerator / device (on a GPU rather than CPU) and teams distribute creates a team of worker threads, where each thread runs on a GPU core, and distributes work amongst them.

Parallel Algorithms

Although we discuss collectives like reduction, broadcast, etc. in the context of scientific calculations in this class, it has applications in fields like AI as well (ex. when training large language models in parallel, we need to use collectives to get the final result). One calculation with wide applications, especially in ML and AI, is matrix multiplication. First we discuss the basic concept of matrix multiplication. We have 2 input arrays A and B and 1 output array C. The number of columns in the first matrix A **must** equal the number of rows in the second matrix B. The resulting array C has the same number of rows as A and the same number of columns as B. Then, to calculate A * B = C, for a cell of C in row i and column j, we take row i of A and column j of B, perform pairwise multiplication, and finally take the sum of those multiplications.

A performance issue that may arise for large arrays is that depending on the language and its model, it may be easier to access one array and have more cache misses for the other array. Assuming that both matrices A and B are stored in row major, in the case of A, an entire row may not fit in cache, and in the case of B, when we try to access a single column, since the column elements are not in contiguous parts of memory, we may need to bring all rows into memory.

We can solve this problem by using blocking to improve cache performance. Blocking is a sequential optimization that can be done before performing multiplication. We create smaller blocks of the matrix that fit in cache, leading to cache reuse (For example if we had a 128x128 matrix, we could split it up into blocks of 32x32). Then, we can bring in one block of A and B at a time into memory, perform matrix multiplication on those blocks, then write the data out. If block sizes are chosen properly, for a single block of A, we can fit an entire row in memory. Furthermore, for a single block of B, to access a single column, we can bring in all rows on B without needing to throw out data in the cache.

Next we discuss the parallelization of matrix multiplication in the context of a distributed memory programming model. We can perform a similar scheme as above to implement parallel matrix

multiplication. Assuming A and B don’t fit in memory, we can store blocks of A and B in distributed memory and communicate between processes to get the right sub-matrices to each process. Each process has a smaller portion of A and B and computes a portion of C. Finally, we reduce across all processes to get the final, complete version of C.

Citations: OpenMP Lecture Slides