Andrew Craze
CSE430 - Syrotiuk
04 November 2014

CSE 430 Project 2

Design & Analysis

Andrew Craze

Directive, Clause and Routine Implementations

For this project, we were assigned the task of writing an OpenMP pre-processor which reads in a multi-threaded OpenMP C/C++ program and then produces a POSIX Pthread implementation of the exact same program. In order to meet these requirements, it was necessary to implement the OpenMP *parallel*, *parallel for*, *critical* and *single* directives using only the POSIX Pthreads library. Also required was the implementation of the OpenMP *num_threads(int), shared(list)* and *private(list)* clauses, as well as the *omp_get_thread_num()* library routine. The implementation of each of these directives, clauses and library routines are described as follows:

#pragma omp parallel

This directive allows the compiler to parallelize a specific code block. Implementing this directive using POSIX Pthreads requires us to create Pthreads in our main function like this:

```
pthread_t thr[NUM_THREADS];
pthread_create(&thr[index], NULL, do_work, &thr_data[z]);
```

where the first parameter, `&thr[index],` is an array of pthreads, which is being sent to the `do_work()` function (3rd parameter). The last parameter (`&thr_data[z]`) is an array of structs which holds data for each of the threads stored inside of the `pthread_t thr[ ]` array. When the threads are passed to the `do_work()` function, the `thread_data` struct is passed to the function as well(4th parameter). A basic `thread_data` struct looks similar to the following:

```
typedef struct _thread_data_t {
  int tid;
     <insert private variables here>
} thread_data_t;
```

where `tid` is the thread id/ thread number of pthread associated with it. When private variables are declared, they are moved inside of this struct.

The `do_work()` function is where the parallel block of code is executed in my pthreads implementation. Once all the threads have finished their concurrent tasks, we join them back together in the main function like this:

```
for (index = 0; index < NUM_THREADS; ++index) {
    pthread_join(thr[index], NULL);
  }
```

Once all of the threads have been joined again, the program resumes executing commands sequentially.

#pragma omp parallel for / #pragma omp for

This directive allows you to define a parallel region of code which can contain a single for loop and the compiler will split iterations evenly amongst the threads. In my implementation, I needed to figure out how to split up the iterations between each thread. My idea was to stagger each thread based on it's thread-id multiplied by a 'chunk_size'. The chunk_size was calculated by dividing the size of the for loop against the total number of threads. These calculations are made for each thread when they enter the *do_work()* function. It looks similar to the following:

```
int tid = data->tid;
int chunk_size = (SIZE / NUM_THREADS);
int start = tid * chunk_size;
int end = start + chunk_size;

for(int i = start; i < end; i++){
   // parallel for loop code…..
}
```

#pragma omp critical

This directive defines a section of code that can only be executed by a single thread at a single time. To implement this construct using POSIX pthreads, I needed to instantiate a `pthread_mutex_t`, as well as use a `mutex_lock` to ensure that only one thread can enter this critical section at a time. The implementation looks similar to the following:

```
pthread_mutex_t var=PTHREAD_MUTEX_INITIALIZER; // in header…

pthread_mutex_lock(&var);  // lock the critical section

sum += data->local_sum;
printf( "Thread %d: local_sum = %d, sum = %d\n", data->tid );

pthread_mutex_unlock(&var); // unlock once you are done
```

#pragma omp single

This directive defines a section of code that can only be run by a single thread. This is useful for non thread-safe calls to I/O or handling portions of code that do not necessarily need to be in parallel. In my implementation I did two things, I set a global boolean flag to false and then inserted the single construct code at the beginning of my parallel function. It checks to see if the global boolean flag is false, and then executes the single construct code. After it has executed this code for the first time, the global boolean flag is set to true, keeping all other threads out of the region of code. The implementation is as follows:

```
bool signalFlag = false; // global in header….

  if(!signalFlag){
    signalFlag = true;
    <single construct code region...>
  }
```

---

num_threads(int) clause

This clause allows the programmer to specify how many threads will be used in the parallel section. In order to implement this with pthreads, I looked for this clauses in each line of the OpenMP program. Once this clause is found, the integer parameter is parsed and then saved as a global variable in the processed program's header. An example of this clause follows:

```
num_threads(4) // this is what we see in OpenMP program

#define NUM_THREADS 4 // what we see in Pthreads program header
```

---

private(list) clause

This clauses specifies which variables are specific to each thread. Each thread will have it's own unique value for each of the variables found in this list. As I had mentioned earlier, we append these variables inside of the thread_data struct to ensure that each thread gets their own copy of the variable. An example of this clause follows:

```
private(foo, bar) // what is seen in OpenMP program

typedef struct _thread_data_t { // pthread data struct
  int tid;
  int foo;
  int bar;
} thread_data_t;
```

---

shared(list) clause

This clause allows member threads to access the variables contained in the list. The way I implemented this was relatively straight forward. When parsing each line of the OpenMP program, when I came across the shared clause, I moved the variables contained in the list parameter to have a global declaration inside of the new pthreads implementation. Then I check for other declarations of those variables and remove them.

---

omp_get_thread_num() library routine

This routine is used to grab the executing thread's identification number. As I had mentioned earlier, each thread's ID is stored inside the thread_data struct. When this routine exists within a line of the OpenMP program, it is converted into data->tid, where data is a pointer to the struct of the current executing thread's data and tid is the thread ID number. The struct, beginning of the do_work function, and call to thread identifier is as follows:

```
typedef struct _thread_data_t {
  int tid;
} thread_data_t;

void *do_work(void *arg) {
  thread_data_t *data = (thread_data_t *)arg;
  printf("Hello from thread %d \n", data->tid);
.....................................................
.....................................................
```

Pre-Processor program design

The pre-processor program reads an input program, which contains OpenMP directives, and outputs a new program where the required behaviour of the OpenMP directives are implemented using the POSIX pthreads library. The output file, **xpost.cc** should have identical output as the input program **x.cc.**

When implementing a multithreaded program using the pthreads library, regardless of what functionality is being implemented, there is a large amount of boilerplate code which I was able to break up into templates. I named the vector<string> templates `header`, `pthreadStruct`, `workFunc`, and `mainFunc`. Each of these templates are filled in as the input program is parsed line-by-line until the end of the input program has been reached. There is also boilerplate

template code for each directive which is hardcoded in the methods `loadParTemplate()`, `loadParForTemplate()`, `loadCriticalTemplate()` and `loadSingleTemplate()`.

The input program is read in once, and each line is parsed then checked for the presence of a #pragma statement. If a #pragma statement is found, we look to find which directives or clauses are being called, along with checking for any library routines as well. A template is loaded based on which directive or clauses are called, then the program enters a 2nd 'read loop' called `_readDirectiveProgram()` which parses the program at the scope level of each directive, then exits back to the main program loop (or the next directive loop). Inside the 2nd directive read loop, I look for additional #pragma statements, as well as any private or shared variables. It should be noted that the pre-processor supports nested omp directives. If another #pragma statement is found, the respective template is loaded and we go into yet another `_readDirectiveProgram()` loop.

Finally, once the entire input program has been parsed and all the template vectors have been filled, we merge all the templates together and form the post-processed file. The method `saveProcessedProgram()` concatenates all of our vector<string> templates and writes them to file inside the `OUTPUT`/ directory.