CSE 430 Project 2

David Ganey

October 31st, 2014

1 Introduction

The task for this assignment was to write a program which takes a C++ program as input and outputs another C++ program with the same functionality. The "translation" program should convert the OpenMP calls in the original program to use the `pthreads` library in the output. There were no limitations on the manner in which this translation must be done (e.g. the file could be read as many times as needed)–the only requirement was compilable parallel code using `pthreads`.

2 Limitations

The assignment initially seemed very difficult due simply to the scope of the project. Some assumptions were required for us to be able to parse the input files, since the possible grammar for C++ is so extensive. For example, the assumption that all private or shared variable types would be primitives (instead of structs or pointer types) helped significantly. Without the assumptions added to the project, a full lexical analysis of the program would have been required prior to conversion, as we would need to build a symbol table in order to know what type each variable was for redeclaration. Of course, some limitations were present in the initial assignment document–specifically the OpenMP constructs which we were required to implement.

3 Methodology

This section will aim to describe at a high level the basic structure of my translation program. From there, more specific details will be given regarding each specific OpenMP construct and how translation of those was achieved.

Fundamentally, the program primarily operates on a large `std::vector<std::string>` structure. One string vector (called `input`) holds the entire program after it is read in. This allows the translator to only read the input program once, and then to operate on a mutable structure held in memory. A drawback of this approach would be its memory consumption for large C++ programs, but for this assignment this was not an issue.

As the program is read in, the strings are trimmed on the left and right to remove whitespace. This allows for assumptions regarding the character index 0 (for example, the ease of recognizing `#pragma` statements by checking for a `#` symbol at 0). Once the entire input program is stored in a vector, the variables are processed. Variable processing is done by checking if the first word of a line is a type (e.g. `int`). Except in the case of "`int main()`", this indicates a variable declaration. The program checks this line for commas (in case of multiple variable declarations) and then stores each variable in a hashmap (`std::map<std::string, std::string>`). This maps from the variable name to its type, and is used later when declaring variables. Additionally, another map (`std::map<std::string, int>`) is used to record the line on which each variable is initially declared.

After processing variables, the main processing begins. Each construct is handled independently, using two functions. A primary function (e.g. `processParallel()`) simply looks for a `#pragma omp parallel` line. It establishes the length of the construct, and passes integers representing the start and end of that construct to a helper function which actually does the processing (see below). The primary function then returns true if it called this helper function, and false if it never found the relevant `#pragma`. Therefore, processing is done as follows:

```
bool foundConstruct = true;
while (foundConstruct)
{
    foundConstruct = processParallel();
}
```

This is repeated for each construct variety. At the very end, it iterates through the entire program looking for `omp_get_thread_num()` calls, and replaces those as well. Finally, it declares global variables and new `include` statements at the top. The last task of the program is to print the entire `input` vector, which will be redirected to the appropriate file on the command line.

3.a Parallel

The helper functions for both `parallel` and `parallel for` clauses are very simple, and are described here. Helper functions analyze the `#pragma` line and record the private and shared variables, as well as the number of threads (which is saved globally). It then creates the new pthreads function. It starts by redeclaring private variables (using the hashmap described above). It copies the function code as-is to the new function. It then uses a loop to dispatch the pthreads.

Before creating each pthread, it creates a StartEnd struct which is passed to the thread. The values for the struct are modified appropriately. The struct contains the following data:

```
struct StartEnd
{
    int start;
    int end;
    int threadNum;
};
```

For standard `parallel` constructs, it simply creates a pthread for each thread. There is no need to fill the "start" or "end" values of the struct, simply the thread number. For `parallel for`, however, it must divide the iterations among the threads.

3.b Critical

The `#pragma omp critical` construct establishes a critical section around an area. This is achieved in this code through the use of a global mutex. The mutex is declared at the top, and when a critical section is found, `pthread_mutex_lock` and `pthread_mutex_unlock` commands are placed around it.

3.c Single

The `#pragma omp single` construct states that only one thread can execute the code contained within. For this assignment, I chose to restrict execution to a specific thread number. The logical choice was thread 0, which is guaranteed to exist if a pthread is running. A simple if statement establishes if that thread is permitted to enter the single section.

4 Performance

This program is not a shining example of efficiency. While it runs very quickly on small inputs, it is unlikely to be successful with large programs due to potentially high memory use. Effort was made to pass vectors by reference (thereby avoiding copies), but this translator should still be restricted to small C++ programs. Additionally, the parser is unlikely to be correct for *all* inputs. It does replicate the OpenMP performance of the sample inputs, but it is easy to tell the program is relatively fragile and is unlikely to succeed on more complicated input files.

5 Execution

The easiest way to observe the program is with the included `run.sh` shell script (below). The script compiles the program and runs it on a C++ file called `input.cc`, redirecting the output to `output.cc`. It then compiles this output program and runs it.

```
g++ main.cpp main.h -std=c++11 -o pj02
./pj02 < input.cc > output.cc
g++ output.cc -lpthread -o out
./out
```

Alternatively, manual compilation can be achieved using the same commands. A Makefile is included for convenience, but only builds the translator program. It does not run it or supply it with any input, nor does it operate on the resulting file.