Programming with OpenMP

Homework 2

Alex Lindberg
alex5@kth.se

February 11, 2022

Introduction

This is a short report detailing the first homework — *Critical Sections*. Implementations for task 1, 2 and 4 are included. All implementations were done in C using the pthread library.

1 Matrix Sum

1.1 Min & Max

The first part consists of computing the minimum and maximum values in a matrix and printing them, as well as initializing all elements to random values.

The values can be randomized by using the `rand()` function modulo some value. `rand()` was seeded using `srand()`.

Finding the min and max was done by setting min and max to the first number in the matrix and updating them if needed when we compute the total value of a row. Using the `for` construct with a reduction clause we can calculate the total value using multiple processors.

```
#pragma omp parallel for reduction(+:total) ...
```

When we reach new potential min/max value we can use the `critical` directive to ensure that only one thread at a time can access the code block. Another solution would be to use a `omp_lock_t` mutex which we lock and unlock before modifying the min/max values.

Performance Evaluation

The difference in execution time can be seen in table 1. The results show that we can gain some speedup for large matrices. Noticeably we can see some performance gain from $1 \rightarrow 4$ threads when evaluating 1000×1000 sized matrices and surely this would continue for even larger ones (given a better choice of data structures). Further increasing the amount of threads however, we see that we gain very little performance. The reason why could probably be that the array still isn't large enough to warrant that many threads. `openmp` still needs to start the thread and divide the indicies into chunks and assign them to the threads, which could negate the performance gain.

n	Threads	Time $[ms]$
10^2	1	0
10^2	2	0
10^2	4	0
10^3	1	1
10^3	4	1
10^3	12	1
10^4	1	60
10^4	4	18
10^4	12	15

Table 1: Execution time of a parallel matrix element sum program.

Quicksort

To complete this assignment I re-used some code from the previous lab. I then removed all the pthread functionality and switched to openmp. We use the `task` directive to recursively create tasks that can be picked up by another thread.

```
#pragma omp task
{
    parallelQuicksort((void *)&lo_args);
}
```

The first call to the quicksort method is done with a `single` directive to ensure only one thread executes.

Performance Evaluation

The results of different input sizes can be seen in table 2. For small arrays we can see that the difference in time is non-existent, likely because the program is too fast to properly analyze. For larger arrays we can start to see a difference in time. 6+ threads is around 4 times faster than a single thread, but when we try to increase from $6 \rightarrow 12$ threads we gain no increase in speed. This could probably be caused by same reason detailed in the previous performance evaluation.

n	Threads	Time $[ms]$
10^2	1	0
10^2	4	0
10^4	1	1
10^4	4	1
10^6	6	15
10^6	12	23
10^7	1	420
10^7	6	112
10^7	12	115

Table 2: Execution time of a parallel quicksort program.

Palindromic Words

The idea for this program was to read every word from `words.txt` into a list of strings (`char[][]`) and using another list of the same size we can mark which words are palindromic. That way we only need to worry about critical sections when updating the palindromic words counter and the marking list.

After constructing the word map and the marker map we can use the `for` construct to iterate and `reduction(+:numWords)` to update the number of palindromic words. Since we never modify the same index on more than a single thread we shouldn't need to mark sections with `#pragma critical`.

```
char word[MAXWORDSIZE];
strcpy(word, wordMap[k]);
reverse(word);
if (strcmp(word, wordMap[k]) == 0 ||
    binarySearch(totalWords, word))
{
    palindromicMap[k] = 1;
```

```
    numWords += 1;
}
```

The `words.txt` file is conveniently already sorted, which means we can use string comparison and binary search to find the potentially matching palindromic word.

Performance Evaluation

As seen in table 3 there is barely any difference for the "words.txt". This could be caused by `omp_get_wtime()` not being able to record a smaller time value, but equally likely that the word list is simply too small or processor too fast. We can however see a some difference in the results when evaluating the full list, where we observe that 12 threads running is about $4x$ faster than a single thread.

Words	Threads	Time [*ms*]
256	1	0
256	4	0
256	8	0
5000	1	1
5000	4	1
5000	8	1
25143	1	4
25143	6	2
25143	12	1

Table 3: Execution time of a parallel palindromic word searcher.