PX425 Assignment 3 (`percolation.c`)

`phuwcs` (1833194)
Department of Physics, University of Warwick
(Dated: November 26, 2021)

Exploration of different ways of parallelising a program using OpenMP.

CONTENTS

I. INTIAL RUN

Running the unmodified code produced the following output:

```
Pcon =          0.0004 Av. Num. Clusters. =      816.0000 Av. Largest Cluster =     3995.0000
Time elapsed :         0.1008 seconds
```

II. THE DATA RACE

Since the initial loop is parallelised over `i`, each thread will be working with its own i. However, j varies from `i + 1` to `Nvert - 1` on each thread, creating a possible data race. This was on lines `136`, `139`, and `145`[1] of the unmodified code. Line `136` calculates a new value for `Ncon[j]`, and lines `139` and `145` both access that value. This would cause the value of `Ncon[j]` to be smaller than it should if two threads write to it at the same time. It could also cause a segmentation fault if the value of `Ncon[j]` increases beyond the bounds of the `Lcon` array between the check and the code that edits `Lcon`.

I think the ideal way to fix this would be to calculate the value of `Ncon[j]` once and then use that calculated value throughout the loop:

```
#pragma omp atomic
int Nconj = (Ncon[j] += 1);
```

However since the script asked only for using OpenMP directives, I enclosed the contents of the `if` statement body with a `#pragma omp critical`. Since Pcon is typically small, the probability of two threads running this region at the same time is small but it did still result in a small slowdown of approximately one second.

By changing the scheduling strategy I was able to get different results, but this is probably be because the way random number generator is initialised will result in slight variations when the scheduling is changed and different threads generate `xi` for different i, `j` pairs. In order to check, I modified the code as such:

```
long nconj = (Ncon[j] = Ncon[j] + 1);

// Check that we will not overrun the end of the Lcon array
if ((Ncon[i] > Maxcon - 1) || (Ncon[j] > Maxcon - 1)) {
    printf("Error generating random graph.\n");
    printf("Maximum number of edges per vertex exceeded!\n");
    exit(EXIT_FAILURE);
```

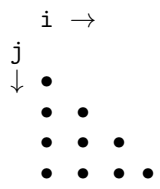
```
}
// j is connected to i
Lcon[Maxcon * i + Ncon[i] - 1] = j;
// i is connected to j
Lcon[Maxcon * j + Ncon[j] - 1] = i;

if (nconj != Ncon[j]) {
    printf("Data race actually happened!\n");
    exit(EXIT_FAILURE);
}
```

Which confirmed that this was happening occasionally. I assume that it was not detectable in the results because of the precision outputted. After the `critical` region was added, no runs resulted in data races being caught by this guard. Since the script did not ask for this being added, I have omitted it from `percolation1.c`.

III. PARALLELISM OVER VERTICES

With the data race out of the way, I moved on to tweaking scheduling. The base time for `Np = 1`, `Ngraphs = 1`, and a chunk size of `Nvert / 4` was around 0.05s on `stan1`. Setting `Np = 8` and `Ngraphs = 20` with the same chunk size lead to a fairly consistent 29s execution time.

When considering a faster chunk size, I thought about the amount of work performed in a single iteration. Since `j` varies from `i + 1` to `Nvert - 1`, the smaller the value of `i` the greater the work that has to be performed. The amount of work that has to be done for a given value for `i` forms a triangle:



where each dot represents a unit of work. In order to balance the largest bits of work, when `i` is small, a small chunk size should be used.

Chunk size	`Nvert` equivalent	Time / s
1	`Nvert / 5000`	15.
2	`Nvert / 2500`	16.
3	`Nvert / 1666`	16.
10	`Nvert / 500`	16.
100	`Nvert / 50`	16.
1000	`Nvert / 5`	17.
5000	`Nvert / 1`	23.

Table I. Execution time as the chunk size varies, using static scheduling. `Np = 8, Ngraphs = 20, Nvert = 5000`.

My findings appear to confirm my reasoning. For comparison, I also tried using `dynamic` scheduling with a chunk size of 1, however this was slower by about 2s than its `static` counterpart. I suspect this is because of the overhead added by the runtime in allocating work — since the amount of work performed by each `i`, `j` pair is effectively pre-determined, it is no better than the `static` allocation where all the expensive work is shared out round-robin.

IV. PARALLELISM OVER `P`

The existing `omp parallel` directive was removed and replaced with `#pragma omp parallel for default(none)` on the `ip for` loop. Table II shows the data sharing clauses that were used for each variable needed by the loop.

Variable	Data Sharing Clause
Pcon	private
igraph	
Ncon	
Lcon	
i	
j	
xi	
lclus	
nclus	
avlclus	
avnclus	
Np	shared
Ngraphs	
Nvert	
Maxcon	
Pcon_step	

Table II. Data sharing clauses for each variable used in the `omp parallel for` directive.

OMP_NUM_THREADS	Time / s
1	26.
2	17.
3	10.
4	9.5

Table III. Execution time by number of threads when parallelising over `P`.

Table III shows that this is a more effective strategy for parallelism than parallelisation over vertices — the same number of threads produced a signficantly faster time.

In order to make the output into the correct order using an OpenMP directive, the loop was modified to be `omp parallel for ordered` and then `#pragma omp ordered` was added immediately before the `printf` function call. The results of this change are show in table IV.

OMP_NUM_THREADS	Time / s
1	27.
2	23.
3	20.
4	16.

Table IV. Excecution time by number of threads when parallelising over P and using `ordered` to ensure correct order of output.

Annoyingly this resulted in a significant performance penalty due to threads being held up while waiting for others to complete their work and print results. A more efficient approach to this is discussed in VI on the following page.

V. PARALLELISM OVER GRAPHS

Now a `#pragma omp parallel for default(none)` was added to the loop over `Ngraphs` instead of the loop over `Np`. Table V shows the data sharing clauses that were used for each variable needed by the loop.

Variable	Data Sharing Clause
`Pcon`	`private`
Lcon	
i	
`j`	
`xi`	
`lclus`	
`nclus`	
`Ngraphs`	`shared`
`Nvert`	
`Maxcon`	
`Pcon`	
`avlclus`	`reduction(+)`
`avnclus`	

Table V. Data sharing clauses for each variable used in the `omp parallel for` directive.

`OMP_NUM_THREADS`	Time / s
1	26.
2	13.
3	9.3
4	7.2

Table VI. Execution time by number of threads when parallelising over `P`.

This approach proved to be even faster than parallelising over `P`. I think this is because by parallelising over the outer loop there is less forking and joining of threads, which uses less CPU time and also means there are fewer cache misses as the program has to synchronise shared variables across threads.

VI. NESTED PARALLELISM

Combining the two previous approaches was fairly simple. Both `omp parallel for` directives from (IV) and (V) were used, with only minor changes to the data sharing clauses used as described in table VII, and environment variables to control branching are in table VIII.

An additional change was made to the program to ensure output is in correct order. Instead of using an `omp ordered` directive which would slow down results, results are saved to an array called `results` in the correct place. Once the computation is complete, the array is printed out in the correct order without holding up any of the team of theads.

With `Np = 2`, `Ngraphs = 20`, and the previously configured environment variables limiting the maximum number of threads at once to 4, the program runs in approximately 1.3 seconds.

---

[1] In `percolation1.c` these are lines `136`, `139`, and `148` respectively.

Variable	Data Sharing Clause
Pcon	private
avlclus	
avnclus	
Np	shared
Pcon_step	
Ngraphs	
Nvert	
Maxcon	
results	

Variable	Data Sharing Clause
Ncon	private
Lcon	
i	
j	
xi	
lclus	
nclus	
Ngraphs	shared
Nvert	
Maxcon	
Pcon	
avlclus	reduction(+)
avnclus	

Table VII. Data sharing clauses for both the outer (over graphs, left) and inner (over P, right) loops.

Environment Variable	Value
OMP_NUM_THREADS	2,2
OMP_NESTED	TRUE
OMP_MAX_ACTIVE_LEVELS	2

Table VIII. Environment variables used for the nested parallelism case.