Lyon Kee

keel@oregonstate.edu

Project #3: Parallel Programming Challenge

Key snippets of code

```
107     #pragma omp parallel for default(none), shared(Cities, Capitals)
108     for( int i = 0; i < NUMCITIES; i++ )
109     {
110         int capitalnumber = -1;
111         float mindistance = 1.e+37;
112
113         for( int k = 0; k < NUMCAPITALS; k++ )
114         {
115             float dist = Distance( i, k );
116             if( dist < mindistance )
117             {
118                 Cities[i].capitalnumber = k;
119                 mindistance = dist;
120             }
121         }
122
123         int k = Cities[i].capitalnumber;
124         // this is here for the same reason as the Trapezoid noteset uses it:
125         #pragma omp critical
126         {
127             Capitals[k].longsum += Cities[i].longitude;
128             Capitals[k].latsum  += Cities[i].latitude;
129             Capitals[k].numsum++;
130         }
131     }
```

Line: 107: We specified Cities and Capitals as shared variables between threads. Cities are used between threads in lines 118, 123, 127, and 128. Capitals are used in lines 127 to 129 to sum value to calculate the average longitude and latitude as the step two algorithm for k-means algorithm. As seen in the diagram above, there are little parallelizable code in the entire code that is being ran, the code becomes more parallelizable as NUMCAPITALS increase as the number of cities is fixed across all runs.

Tables of data

Columns: # Threads

Rows: # Capitals

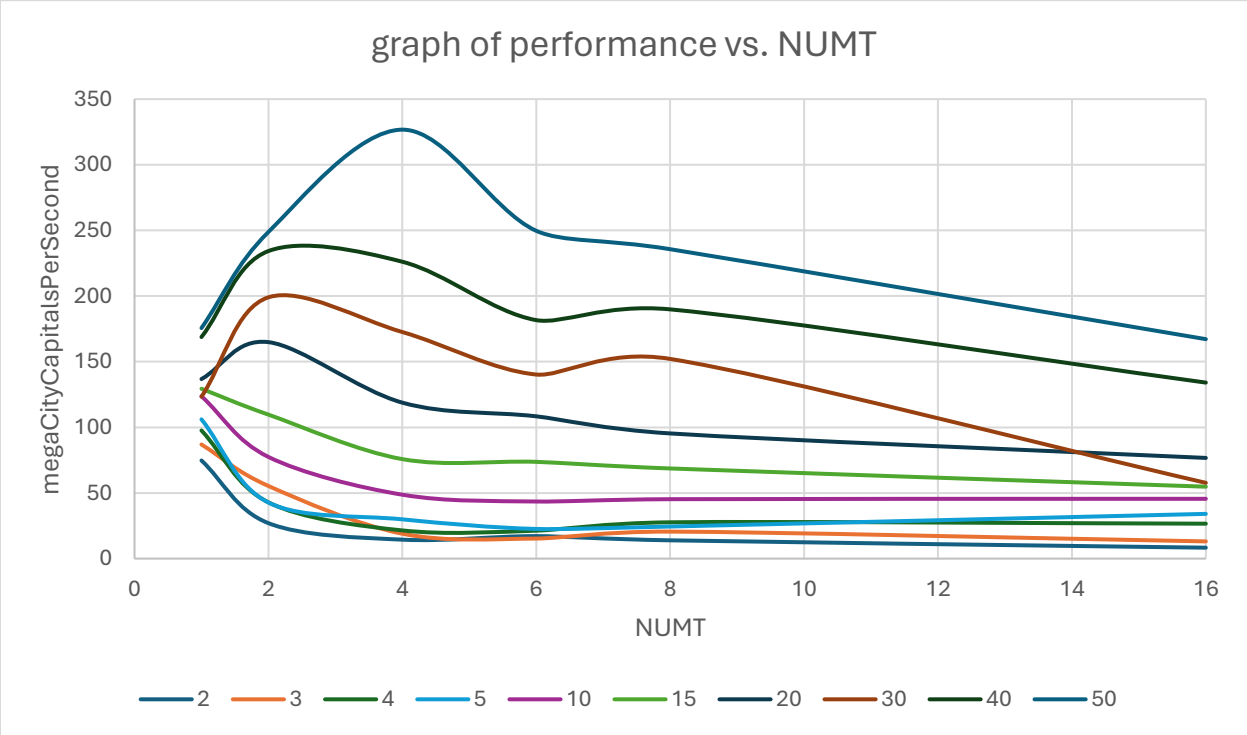
Values: megaCityCapitalsPerSecond

megaCityCapitalsPerSecond	NUMT					
NumCapitals	1	2	4	6	8	16
2	74.768	27.009	14.431	17.131	13.927	8.312
3	86.922	55.32	18.961	15.295	20.57	13.124
4	97.626	42.824	21.469	21.294	27.658	26.606
5	106.09	42.721	29.91	22.607	24.457	34.086
10	123.674	77.447	48.671	43.498	45.259	45.559
15	129.374	109.678	75.767	73.702	68.66	54.797

20	136.78	164.886	118.791	108.375	95.355	76.696
30	123.349	199.026	172.579	140.207	152.109	57.726
40	168.789	234.27	226.12	181.644	189.856	134.075
50	175.612	248.782	326.726	249.664	235.664	167.162

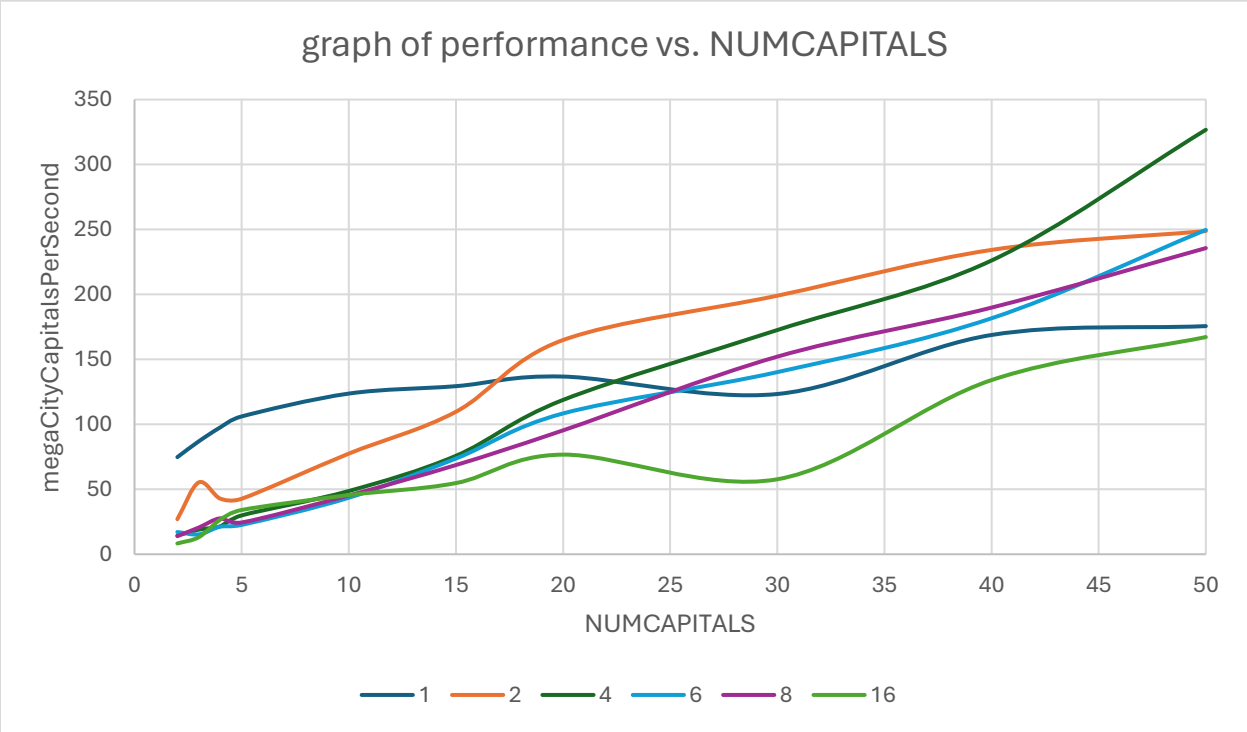
Graphs of data

graph of performance vs. NUMT with the colored curves being NUMCAPITALS



The above graph portrays the trend of performance vs number of threads, it is seen that a higher number of threads is not always be best, as there is a sweet spot and because I'm running on my own laptop, there are many background apps that are interfering with the scheduling and leading to more downtime and also time wasted. It is seen that 4 threads are the sweet spot for many runs, and 1 thread 1-2 threads are the best for runs below 30 capitals, the threads are always split up to run 331 different tasks and the load/ parallelizable fraction increases as the number of capitals increase. It can be said that the load of each thread is little therefore a high number of threads will have a lot of overhead costs which the increase of threads would not be able to overcome and cover leading to 4 threads being the ideal number of threads.

graph of performance vs. NUMCAPITALS cities with the colored curves being NUMT



In this chart we observe a rising trend as the number of capitals increase, the performance also increases. This is because the computation is O(cities * capitals) which meant that each thread will have more computation and the program will have a larger parallel fraction when the number of capitals increase. As the task for each thread increases, the overhead cost will remain he same making the overall downtime take a smaller fraction of the whole computation time, allowing for more processing time than overhead time, thus a higher performance.

Conclusion

1. Tell what machine you ran this on

This is my own laptop with 16 available threads

2. Tell what operating system you were using

I am using the windows operating system with WSL

3. Tell what compiler you used

g++ with -lm and -fopenmp tags

4. Include the table of performance data.

Included above in the **Tables of data** section

5. Include a graph of performance vs. NUMT with the colored curves being NUMCAPITALS.

Included above in the **Graphs of data** section

6. Include a graph of performance vs. NUMCAPITALS cities with the colored curves being NUMT.

Included above in the **Graphs of data** section

7. Tell us what you discovered by doing this. What patterns are you seeing in the graphs?

The pattern that are found seems weird because of the rise and falls when number of threads increases, however further investigation and thinking has led me to think more about the parallelizable fraction and how much computation is needed by each thread. This then allows us to deduce that the weird trend is caused by both a sweet spot of threads that is found on every system and also a small computation that is needed by every thread.

How did things turn out?

This turned out pretty well, I had a small mistake because I did not read the structs beforehand and assumed some things. And as always, on my own laptop, I need to change the number of threads to experiment on because I have 16 threads and the behavior is slightly different from a server due to the scheduling of OS and other applications in the background.

Why do you think they turned out that way?

The charts turned out the way that it is, is due to the nature of program and also scheduling, there is always a cost to starting threads, and if each thread's responsibility is not a lot then it is not worth it to create a thread for it. Another thing is that if each task if too large, then work load is not distributed evenly as the final task might take a very long time to finish, and at the same time, all other threads are doing nothing when the last task can be broken up and be done by other threads.

NOTE: I did extra credit, all statements prints to stderr except for results so we can use `>` to send stdout to another file and not other logging messages.