CS4234
Project 3

Emil Bengtsson

November 2018

Introduction

This project aims to compare how different orderings of for loops can affect parallel performance, and to discuss how OpenMP compares to pthreads and MPI as a way to write a parallel program.

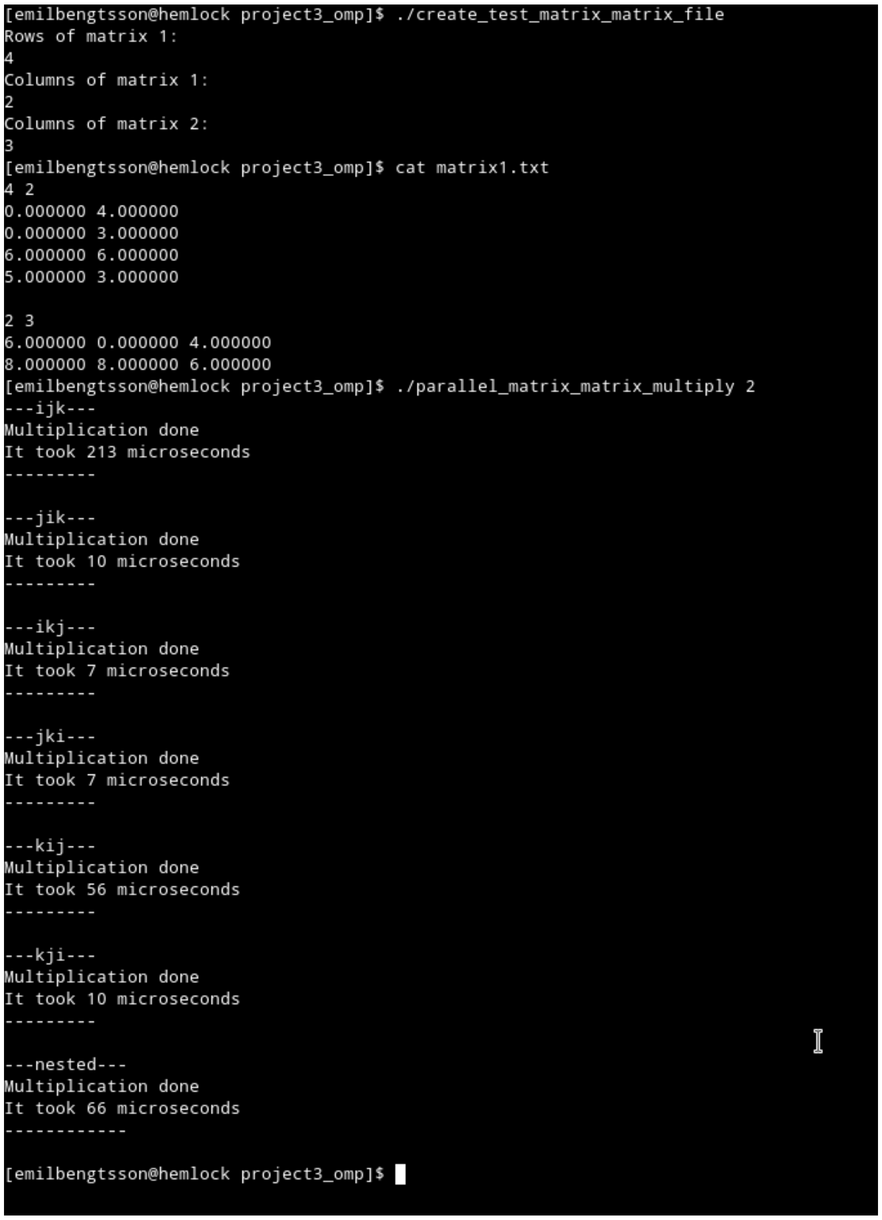
Implementation

For the first task of the project I was supposed to implement matrix-matrix multiplication using loops, and then parallelize them in different orderings. I decided to implement each ordering in its own function so I could switch between orderings easily. The second task for the project was to implement the same matrix-matrix multiplication, but instead of only parallelizing the outer loop both the inner and the outer loop was to be parallelized. I implemented this in a function too, by using the "collapse" clause. The collapse clause "collapses" a number of consecutive loops into one loop that then gets parallelized, which means that if I use it on the two outer loops they will both be parallelized. When I had all the functions I decided that I might as well run all of them at the same time, so the program runs all the different functions and measures the time it takes for each function to complete.

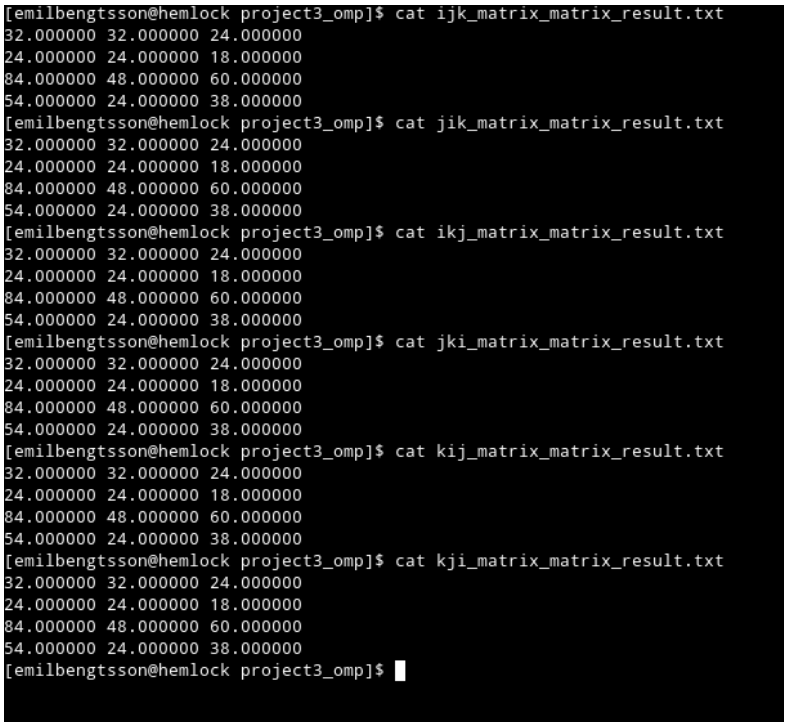
Screen shots of program

This section contains screen shots of correct execution of the program.

Matrix-Matrix Multiply



```
[emilbengtsson@hemlock project3_omp]$ ./create_test_matrix_matrix_file
Rows of matrix 1:
4
Columns of matrix 1:
2
Columns of matrix 2:
3
[emilbengtsson@hemlock project3_omp]$ cat matrix1.txt
4 2
0.000000 4.000000
0.000000 3.000000
6.000000 6.000000
5.000000 3.000000

2 3
6.000000 0.000000 4.000000
8.000000 8.000000 6.000000
[emilbengtsson@hemlock project3_omp]$ ./parallel_matrix_matrix_multiply 2
---ijk---
Multiplication done
It took 213 microseconds
---------

---jik---
Multiplication done
It took 10 microseconds
---------

---ikj---
Multiplication done
It took 7 microseconds
---------

---jki---
Multiplication done
It took 7 microseconds
---------

---kij---
Multiplication done
It took 56 microseconds
---------

---kji---
Multiplication done
It took 10 microseconds
---------

---nested---
Multiplication done
It took 66 microseconds
------------

[emilbengtsson@hemlock project3_omp]$
```



```
[emilbengtsson@hemlock project3_omp]$ cat ijk_matrix_matrix_result.txt
32.000000 32.000000 24.000000
24.000000 24.000000 18.000000
84.000000 48.000000 60.000000
54.000000 24.000000 38.000000
[emilbengtsson@hemlock project3_omp]$ cat jik_matrix_matrix_result.txt
32.000000 32.000000 24.000000
24.000000 24.000000 18.000000
84.000000 48.000000 60.000000
54.000000 24.000000 38.000000
[emilbengtsson@hemlock project3_omp]$ cat ikj_matrix_matrix_result.txt
32.000000 32.000000 24.000000
24.000000 24.000000 18.000000
84.000000 48.000000 60.000000
54.000000 24.000000 38.000000
[emilbengtsson@hemlock project3_omp]$ cat jki_matrix_matrix_result.txt
32.000000 32.000000 24.000000
24.000000 24.000000 18.000000
84.000000 48.000000 60.000000
54.000000 24.000000 38.000000
[emilbengtsson@hemlock project3_omp]$ cat kij_matrix_matrix_result.txt
32.000000 32.000000 24.000000
24.000000 24.000000 18.000000
84.000000 48.000000 60.000000
54.000000 24.000000 38.000000
[emilbengtsson@hemlock project3_omp]$ cat kji_matrix_matrix_result.txt
32.000000 32.000000 24.000000
24.000000 24.000000 18.000000
84.000000 48.000000 60.000000
54.000000 24.000000 38.000000
[emilbengtsson@hemlock project3_omp]$ █
```

Results

The numbers in the top line of the tables represent the ordering of the nested loops, or whether two nested loop levels were parallelized. The numbers in the leftmost line of the tables represent the number of threads that are used to run the algorithms. The size of the matrices used is 960x960.

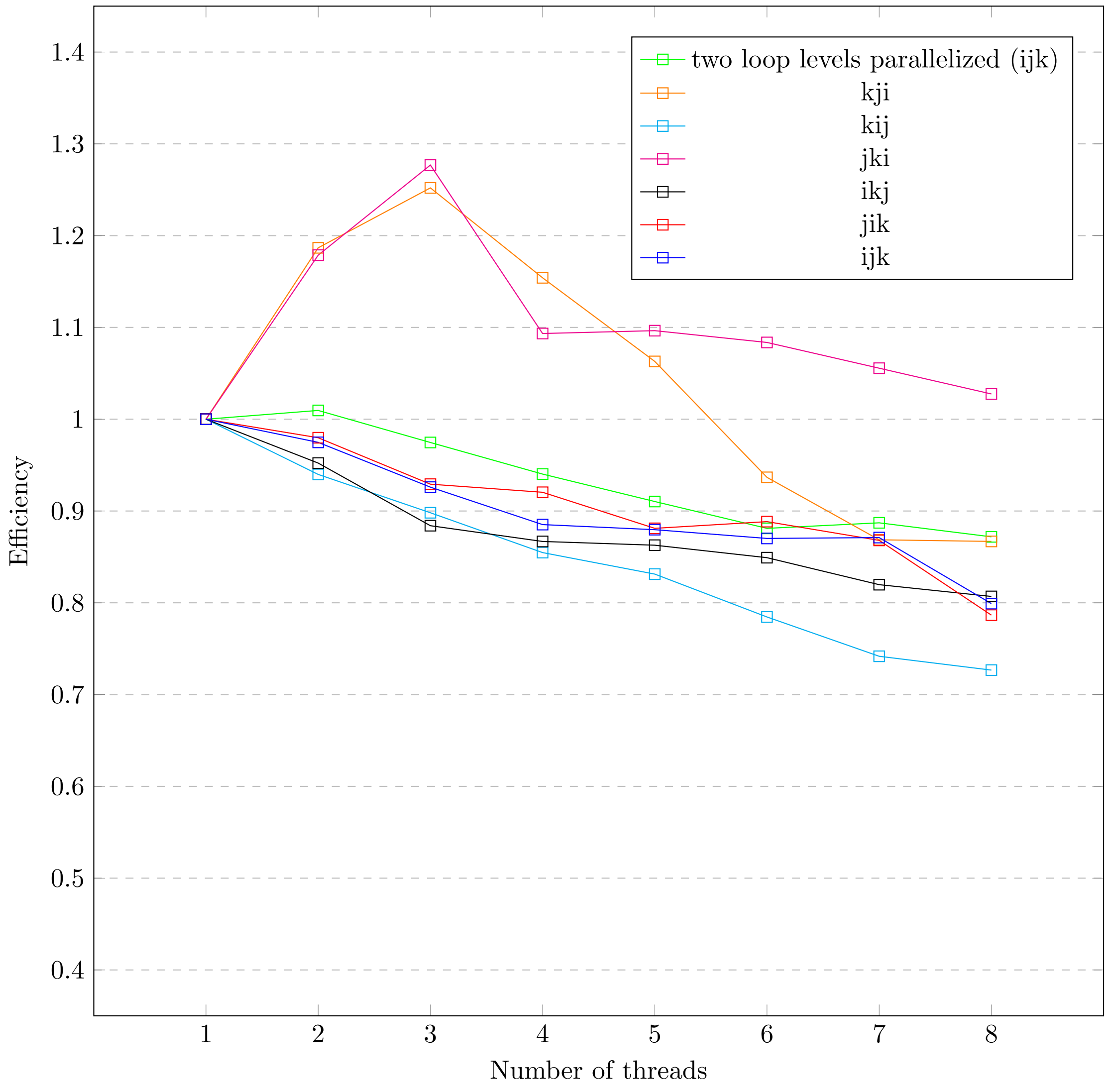
Table of run times (in microseconds)

	ijk	jik	ikj	jki	kij	kji	two loop levels parallelized (ijk)
1 thread	7629568	7612014	4132294	9961395	4160165	10013865	7793533
2 threads	3913486	3884388	2169870	4225528	2213240	4219129	3859822
3 threads	2746460	2730583	1558183	2600459	1544051	2666032	2665421
4 threads	2154768	2067567	1191795	2277678	1216844	2169273	2072258
5 threads	1734655	1727802	957945	1816994	1000928	1884161	1712217
6 threads	1461396	1427987	811066	1532120	883753	1781816	1474212
7 threads	1251282	1252803	720144	1348138	801241	1646894	1255017
8 threads	1193397	1209662	640124	1211829	715612	1443986	1117350

Table of efficiencies

	ijk	jik	ikj	jki	kij	kji	two loop levels parallelized (ijk)
1 thread	1	1	1	1	1	1	1
2 threads	0.97478	0.97982	0.95220	1.17872	0.93984	1.18672	1.00957
3 threads	0.92599	0.92923	0.88400	1.27688	0.89811	1.25203	0.97465
4 threads	0.88520	0.92041	0.86682	1.09337	0.85470	1.15406	0.94022
5 threads	0.87966	0.88112	0.86274	1.09647	0.83126	1.06295	0.91034
6 threads	0.87012	0.88843	0.84915	1.08362	0.78456	0.93667	0.88110
7 threads	0.87106	0.86800	0.81974	1.05557	0.74174	0.86864	0.88713
8 threads	0.79914	0.78658	0.80693	1.02752	0.72668	0.86686	0.87188

Graph of efficiencies for matrix-matrix multiply for different numbers of threads



Conclusion

The first task was to parallelize the matrix-matrix multiplication using the parallel loop in OpenMP, for all six different loop orderings. Some of the loop orderings performed significantly better than others. For example: both ikj and kij were nearly twice as fast as all the other loop orderings. I think this is because these orderings were better at utilizing the cache and got more cache hits, thus speeding up the program. The jki and kji loop orderings had a superlinear speedup and got almost 1.3 efficiency with three threads. I think this is due to them being able to fit more of the loop in the cache as the number of iterations per thread gets smaller, which would result in more cache hits and a faster program.

Using OpenMP to write parallel programs is different from writing parallel programs using distributed memory programming. When you use for example MPI to write parallel programs, if you want multiple processes to have access to some data you have to send the actual data to them. If you use pthreads or OpenMP you use global variables to communicate, since the memory addresses are shared.

The second task was to parallelize two levels of the nested for loops in the algorithm. I did this by using the OpenMP collapse clause, which makes a number of nested loops run in parallel. The threads all get created at the start of the "# pragma omp parallel" and, since the two outer loops were collapsed in my program, each thread can run any row times column combination (as opposed to in the first task where each row is assigned to a thread that then carries out all the row times column operations for that row).

The efficiency of the nested loop was interesting because it declined slower than the efficiencies for the other algorithms, as seen in the graph in the results section. I think this might be because if you parallelize both outer loops, a larger part of the program can be parallelized than if you only parallelize the outermost one.

OpenMP is easier to use than MPI and pthreads in my opinion, and it is very convenient if you want to parallelize a for loop.