Comp 422 Assignment 2 Documentation

The script I wrote for this assignment is solving a group of linear equations (Ax = b) by the method of gaussian eliminations. I would like to illustrate the procedures of what my serial version of codes would do:

The first step that my code does is to setup the matrix A with the given number of rows (Nrow) and number of columns (Ncol). It then fills the matrix A with random numbers by the function drand48() in C. The code does the same things for x and b.

The second step is to do the Gaussian Elimination to transform matrix A into the upper triangular matrix, after the Gaussian elimination, both A and x are in the echelon form.

The third step is to do the back substitution to obtain the final solution of x.

Finally, the code does the final checking of the solution by calculating L2-Norm of (Ax-b), if the L2 Norm is close to zero then our solution is correct.

In the following section I would like to describe the parallization scheme using OpenMP.

There are a couple parts of the program that we can choose to parallize, including Gaussian Elimination part and back substitution part. In my program the Gaussian Elimination part contains several loops, including the parts to do partial pivoting, swapping lines and doing the elimination. The first two parts are inter-independent and fine-grained so it is difficult to parallize. I chose to parallize the loops for doing the elimination, which has the significant speed-ups in terms of the running time of the program. For the back substitution part, because all the steps are fine-grained, so parallization does not really speed up the program. Actually from our test, parallizing the back substitution part slows down the program.

We use the default static schedule in the OpenMP, a relative large chunk size n= Nrow/ p is guaranteed, where p is the number of processors available.

We start to set up the matrix and use the thread binding to memory as provided by the command:

Numactl-localalloc-physcpubind

In my program I distributed the matrix to different threads by using the “omp for” pragma. By doing this we can guarantee the locality of the matrix for the processes within the program.

Finally, we are able to synchronize the parallel work with the implicit barrier in the end of the OpenMP for loop.

In the following section I would like to describe the parallization scheme using Pthread:

Compared with OpemMP, in Pthread we have more control over the distribution among threads. In my program I mainly transform the OpenMP code into the Pthread code, besides that, I also considered the false sharing of the specific line in the given matrix. We did the chunk size distribution for matrix A, and we also assign the matrix into different threads for memory binding convenience in order to improve efficiency.

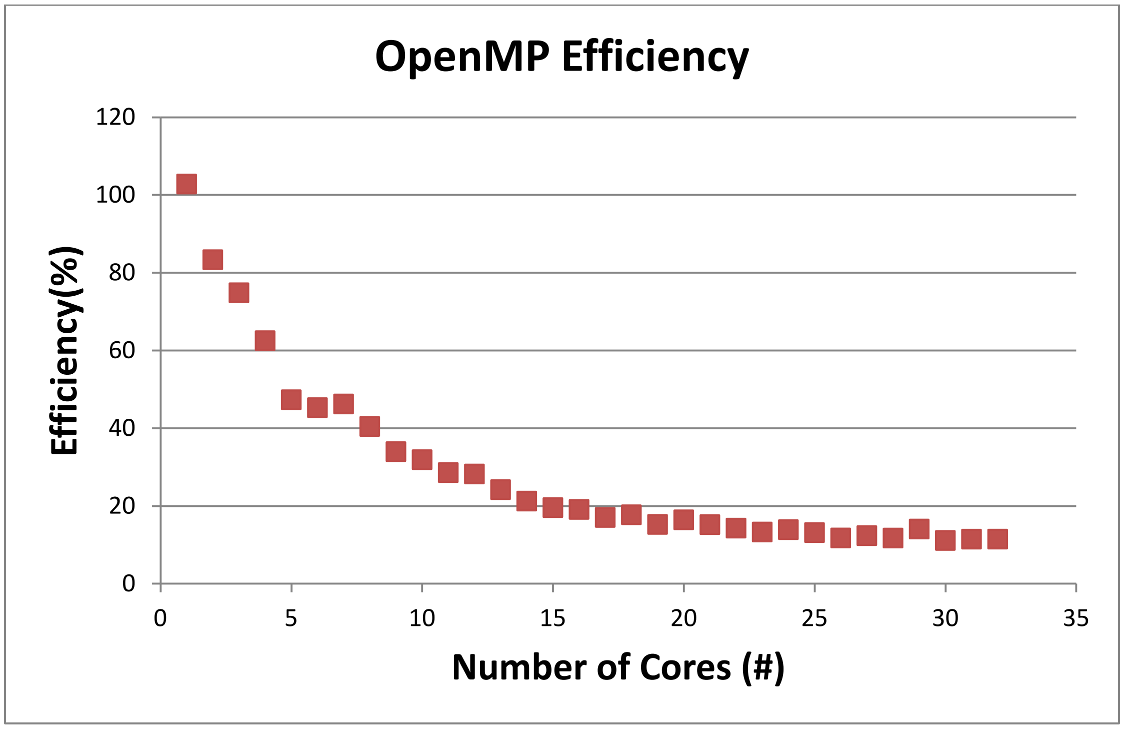
The data of the matrix A and vector b are being read from time to time, so there might be a false sharing problem. In the first loop, we intentionally pick a line (the pivoting line) where the first none-zero value is the smallest. It is highly likely that the data in this line along with the corresponding value in the vector b be read tons of times in the elimination part. In our program I provide a solution to this problem to ensure the locality of the pivoting line of matrix A and the corresponding vector b. The solution is that I copied the pivoting line into different threads in order to guarantee the locality of the frequently-visited cache space. Also, the data is located on the same space with the computation in this way.

In the end, we synchronize the parallel work by the pthread_join() function in C.

The result of Efficiency for OpenMP is shown below:

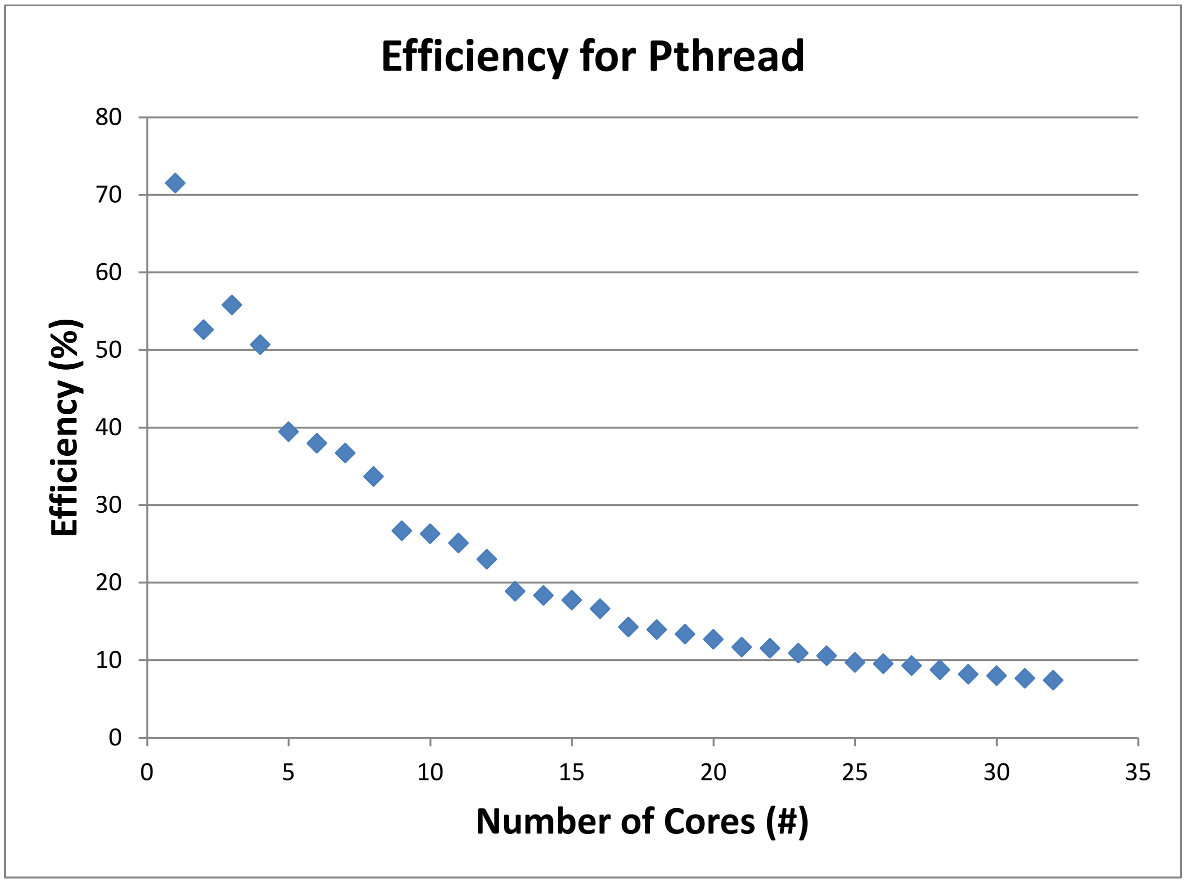
Number of Cores	Times (s)	Efficiency
Serial	221	Serial/(p*T(p))
1	214	102.8038
2	132	83.3333
3	98	74.8299
4	88	62.5000
5	93	47.3118
6	81	45.2675
7	68	46.2185
8	68	40.4412
9	72	33.9506
10	69	31.8841
11	70	28.5714
12	65	28.2051
13	70	24.1758
14	74	21.2355
15	75	19.5556
16	72	19.0972
17	76	17.0279
18	69	17.7134
19	76	15.2354
20	67	16.4179

21	69	15.1829
22	70	14.2851
23	72	13.2850
24	66	13.8889
25	67	13.1343
26	72	11.7521
27	66	12.3457
28	67	11.7271
29	54	14.0485
30	66	11.1111
31	62	11.4464
32	60	11.4583



The results for Efficiency of Pthread is shown below:

Number of Cores	Times (s)	Efficiency
Serial	221	Serial/(p*T(p))
1	309	71.5210
2	210	52.6190
3	132	55.8081
4	109	50.6881
5	112	39.4643
6	97	37.9725
7	86	36.7110
8	82	33.6890
9	92	26.6908
10	84	26.3095
11	80	25.1136
12	80	23.0208
13	90	18.8889
14	86	18.3555
15	83	17.7510
16	83	16.6416
17	91	14.2857
18	88	13.9520
19	87	13.3696
20	87	12.7011
21	90	11.6931
22	87	11.5465
23	88	10.9190
24	87	10.5843
25	91	9.7143
26	89	9.5506
27	88	9.3013
28	90	8.7698
29	93	8.1943
30	92	8.0072
31	93	7.6656
32	93	7.4261



From the results above, we can find that the efficiency for both OpenMP and Pthread decreases when the number of cores increases. The reason for it is that when the number of cores increases, the overhead would be generated. Also the operations of functions in C such as pthread_create() and pthread_join() would take some time and would add additional overhead to the total running time of the program.

Also from the above results, we can find that generally OpenMP is better than Pthread in terms of Efficiency. Because I did not choose to implement the memory binding locality for Pthread. But we know that in Pthread the user has more control over the distribution among threads, I should try to implement the memory binding locality for Pthread if I have more time and in the future.