Introduction to Parallel Programming: Assignment 1

Mayank Singh Kushwaha(2017CS50413) , Navneel Mandal(2017CS50414)

January 24th, 2020

1 Introduction

In this assignment we have to write two shared-memory parallel programs that perform LU decomposition using row pivoting. We will develop one solution using the Pthreads programming model and one using OpenMP.

2 Parallelization Strategy

code for Sequential program.

```
inputs: a(n,n)
outputs: P'(n), l(n,n), and u(n,n)

initialize P' as a vector of length n
initialize u as an n x n matrix with 0s below the diagonal
initialize l as an n x n matrix with 1s on the diagonal and 0s above the diagonal

for i = 1 to n
  P'[i] = i
for k = 1 to n
  max = 0
  for i = k to n
    if max < |a(i,k)|
      max = |a(i,k)|
      k' = i

  if max == 0
    error (singular matrix)

  swap p[k] and p[k']
  swap a(k,:) and a(k',:)
  swap l(k,1:k-1) and l(k',1:k-1)

  u(k,k) = a(k,k)
  for i = k+1 to n
    l(i,k) = a(i,k)/u(k,k)
    u(k,i) = a(k,i)

  for i = k+1 to n
    for j = k+1 to n
```

```
a(i,j) = a(i,j) - l(i,k)*u(k,j)
```

Here, the vector P' is a compact representation of a permutation matrix p(n,n), which is very sparse. For the i^{th} row of p, P'(i) stores the column index of the sole position that contains a 1.

Time taken by sequential code is 1061.643632.

2.1 OpenMP

We are focusing on Task Parallelism for this assignment.
Task Parallelism : Task parallelism employs the decomposition of a task into sub-tasks and then allocating each of the sub-tasks for execution. The processors perform execution of sub tasks concurrently.

2.2 Areas of Parallelism

- **Initialisation :** Initialising the vectors L, U and A. Each element are initialised independently hence this operation can also be initialised.
- **Finding the maximum in a array:** During finding the maximum element in an array we are not changing/updating the elements of the array we are only reading the elements from the array so it is easy to divide the work among the threads keeping *max* and *k'* as shared variable.
- **Swapping :** In swapping two successive operations on the loops are independent of each other also for keeping False Sharing to occur we have kept chunk size as 16 and schedule type as static. Also swap $l(k, 1 : k-1)$ and $l(k', 1 : k-1)$ is the only swapping we need to do parallelly as other two swapping takes O(1) time.
- **Updating Matrix L and U :** While updating the matrix L and U in the loop, the updating operation incrementally takes place. Also **nowait** statement is placed on the **swapping** loop this allows the thread running in parallel on previous **for loop** to continue by breaking the implicit barrier put by the **# pragma omp for**. Motivation behind doing this is that the values that these two loop access is mutually exhausted. **Swap loop** takes value from 0-k-1 and **Updating loop** takes value from k+1-n.
- **Updating Matrix A:** Similar to the above point updating values in the matrix **A** is also independent of each other. OpenMP provides a unique way to parallelise nested loops.

2.3 Partitioning the data

We have majorly used **# pragma omp parallel for** for making our loop parallel. Also, we can see from the pseudo-code and previous subsection that loops operation accesses only single element from any of the Matrix A, L or U which means that last update/access of the data does not affect the overall result of our algorithm. Also, I have defined the chunk-size as 16 which reduces the chance of False sharing because most of the cache line has a size of 32, 64, 128 bytes.

2.4 Justifying Program Sync

Program Synchronisation is mainly maintained by the OpenMP because we have used **# pragma omp parallel for** for parallelizing **for loops**. Only problem that could occur is due to the **nowait** clause used in swapping loop. But as mentioned above in the parallelization strategy the loop for updating the values of matrix L and U which follows the swapping loop does not use any of the data changed by swapping loop. Hence we can see that our program is Synchronised.

2.5 Pthreads Implementation

In this subsection, we will look at how our implementation using Pthreads parallelize the LU decomposition task.

2.5.1 Areas of Parallelism

In the Pthreads implementation, we have parallelised the work at 3 places. In addition to performing the LU decomposition using parallelisation of the final 2 loops after the swapping, we have also created the matrix by parallelising the task.

2.5.2 Generating Random Numbers for each thread

We have 2 different ways of randomizing the numbers generated. We have a **srand** call with the time parameter to randomize the rand() call every time. Now, for each thread we use the random header and use the **Mersenne twister** with the seed **muddled** with **both rand() and the thread id** to generate the pseudo random numbers with seeds different for each thread id.

2.5.3 Partitioning the Data

We define the number of threads and the size of the matrix by getting it as an input from the user. When we are generating the random numbers, We divide the matrix into the number of threads. This is done as follows:

```
Let the size of matrix be n.
Let the number of threads be t.
The ids of the threads go form 0 to n-1.
Now, a thread with id q, will work on matrix with the following rows:
```

$$start\_row = q \times \lfloor \frac{n}{t} \rfloor \tag{1}$$

$$end\_row = (q+1) \times \lfloor \frac{n}{t} \rfloor \tag{2}$$

```
This is true for all but the last thread, which has an additional
work to be done of extra n%t rows.
```

Our implementation tries to make sure that each thread gets a contiguous array. Even the last thread can get a maximum of only (t-1) rows to work upon, which is minimal as compared to the total size of matrix on which we are testing, i.e around 7000-8000.

The above part remains the same for both the random number generation and the LU decomposition. The only difference being during LU decomposition, we don't divide the complete n rows for the t threads, but instead the remaining (n-k-1) rows. thus, the work division changes to :

$$startrow = k + 1 + q \times \lfloor \frac{n-k-1}{t} \rfloor \tag{3}$$

$$end\_row = k + (q+1) \times \lfloor \frac{n-k-1}{t} \rfloor \tag{4}$$

2.5.4 Justifying Program Sync

The above explanation clearly indicates that there can't exist a single row that can be modified by 2 threads of different IDs. We see that during LU Decomposition, there are 2 loops after the 3 swaps.

It is also seen that the 2nd loop requires data from the first loop to be updated, for the correct implementation. So, although we have parallelized both the loops, We have also made sure that on completion of the first parallel part, we duly join the threads, before moving onto the 2nd part. We use this as a good implementation of the fork-and-join model as discussed in class with OpenMP follows.

Defining separate data for each thread also helped us avoid any locks, which could have become the bottleneck for the performance. We had some ideas on even parallelizing the column loop in the 2nd Decomposition loop, but thought that there is good chance of it failing due to Synchronization issues.

3 False Sharing

In symmetric multiprocessor (SMP) systems, each processor has a local cache. The memory system must guarantee cache coherence. False sharing occurs when threads on different processors modify variables that reside on the same cache line. This invalidates the cache line and forces an update, which hurts performance.

Steps taken to avoid false sharing.

1. As we know the cache loads chunk of memory rather than single word. Most common cache line size is 32, 64 or 128 bytes. It makes sense to take chuck-size in openmp as 16 (128/8, sizeof(double) = 8bytes). In pthread we divided the larger loop into n (number of threads) independent loops of almost equal work.
2. We used Column major order in OpenMP program to avoid False sharing in our program.

4 Conclusion

Pthreads:

Number of Threads	Time taken(in sec)	Speedup	Efficiency
1	836.753784	1.269	1.269
2	436.502735	2.432	1.2160
4	256.854246	4.133	1.033
8	316.143415	3.358	0.419
16	300.060989	3.538	0.2211

OpenMP:

Number of Threads	Time taken(in sec)	Speedup	Efficiency
1	1106.626574	0.959	0.959
2	616.610053	1.722	0.861
4	516.672035	2.055	0.5136
8	450.111087	2.358	0.295
16	391.400046	2.712	0.169

We see that on increasing the number of threads, although the speedup is increasing, but the efficiency goes down a lot, showing that the overhead cost of having a lot of threads increases a lot.

In addition, We find that Pthreads give a time minima at 4 threads. This must be due to the presence of 4 cores in the system. We find no such in OpenMP, but this may be due very optimized code in the OpenMP library for higher number of threads.

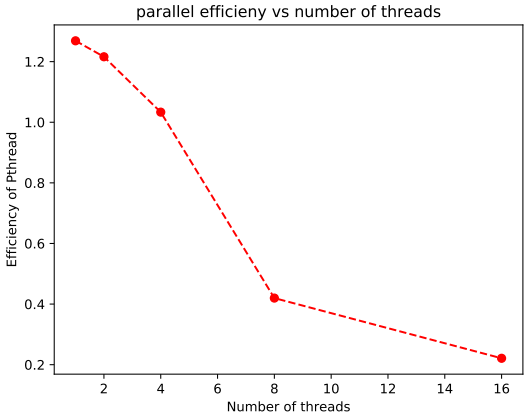

Figure 1: Pthreads Implementation

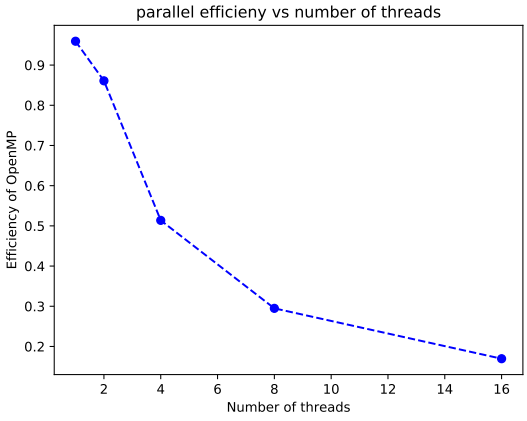

Figure 2: OpenMP Implementation