March 1, 2024

Sequential Program Setup

Top level function

The code is organised into the following chunks for readability and modularity. The main function just calls the (void)top function LU, which is responsible for initialising, decomposing, $L_{2,1}$ computation and timing the decomposition. The signature of the LU function is as follows :

```c
void LU(int n, int t); /* Wrapper function for the decomposition */
```

Initialisers

The following are the signatures of the various helper function that initialise the respective matrices. These matrices are stored as a vector of n pointers to n-element data vectors, for reasons described in further sections.

```c
void init_a(int n, double** m); /* Initialising A with random values */
void init_l(int n, double** m); /* Initialising L with lower triangular values */
void init_u(int n, double** m); /* Initialising U with upper triangular values */
void init_pi(int n, int* m); /* Initialising the output pi with linear integers */
void init_pa(int n, double** m1, double** m2); /* Storing original A as pa, for
    verification of convergence later */
```

Decomposition

After having initialised the matrices, the LU function decomposes the matrix A into L and U as per the following psuedo-code for partial pivoting.

```
for i = 1 to n
  π[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 π[k] and π[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)
```

Timekeepers

The *Chrono* Library is used to clock events and measure the time taken for various operations. These times are then used to infer bottlenecks/false sharing events etc., and optimise them.

The following are the functions used from the library :

```
clock_t t_clock;                         // Declaring a clock
chrono::microseconds t(0);               // Declaring a variable that can store the time
auto t1 = chrono::high_resolution_clock::now();  // Registering an event

auto duration = chrono::duration_cast<chrono::microseconds>(t2-t1); // Computing
    the duration between two clock events
```

Convergence tests

The following signature computes the $L_{2,1}$ norm of the residual matrix $PA-LU$, which is a metric for the accuracy of the above decomposition. Convergence is concluded when the value returned by this function is very low.

```
double L2c1(double** m, int n); // computing the L2,1 norm of the residual matrix
```

The following helpers help in computing the residual

```
void mat_mult(double** m1, double** m2, double** m3, int n); // compute L*U
void mat_rearrange(double** m, int* pi, int n);              // compute PA<- A, pi
void mat_sub(double** m1, double** m2, int n);               // compute PA (-) LU
```

Time

We performed testing and optimisation on 2 systems, we shall refer to them as $S1$ and $S2$ from now on, where $S1$ is our primary device where we could scale the calculations to n = 8000, and $S2$ is our secondary device where we optimised and analysed results for a smaller n = 1000.

$S1$ recorded the sequential time to be **610086** msec for an n=8000 matrix.

$S2$ recorded the sequential time to be **840** msec for an n = 1000 matrix.

OpenMP

The OpenMP code was incrementally improved and optimised over various cycles of observations and attemps to speed them up. The following times are reported for $S2$, on a matrix of n = 1000. We present the analysis of n = 8000 later.

There are 4 main sections to optimise and parallelise code over, run for $k = 1, 2, \ldots n$ iterations :

a) max - Finding the pivot (max) element in the k^{th} column

b) swap - Swapping the k^{th} row and pivot's row for stability in A, and corresponding section in L.

c) lu_upd - Updating the k^{th} column of L and k^{th} row of U after finding the pivot.

d) a_upd - Updating the lower submatrix of A to reflect the computation of the k^{th} column of A.

Note that any optimisation across these different iterations of k is not possible because the of data dependency amongst these loops, where the results of previous iterations are needed to run the next iterations. Thus, the only parallelisation that is possible is within these iterations.

First Draft

To begin with, we naively use *#pragma omp parallel* and the *#pragma omp parallel for* directives in each of the 4 main sections of the decomposition, carrying forth the code from the sequential implementation.

Section	Time	Section	Time
max	1	max	1
swap	3	swap	10
lu_upd	3	lu_upd	7
a_upd	829	a_upd	569
Total	840	Total	592
Efficiency	**1**	Numthreads	**2**

Observations :

a) "a_upd" takes most of the time

b) "max" and "swap" are such small operations that time increases when threads are increased, suggesting that the overhead of parallelisation is perhaps not as rewarding for them. The swaps for the matrix A are simple pointer exchanges, and for swapping rows of L we can use **memcpy**, which allows for data transfer in bulk.

Iteration - I

Intuition :

The ”a_upd” phase consists of the following loop

```
// Outer Loop
for(int i = k+1; i < n; i++){

        // Inner Loop
        for(int j = k+1; j < n; j++){
            a[i][j] = a[i][j] - l[i][k] * u[k][j]; // Notice u[k][j]
        }
}
```

Observe how the value of $u[k][j]$ remains constant throughout the execution of the inner loop! Instead of making repeated memory access calls, we can set this parameter to constant. After converting $u[k][j]->u\_k[j]$ by

```
// Outer Loop
for(int i = k+1; i < n; i++){
        double* u_k = u[k];
        // Inner Loop
        for(int j = k+1; j < n; j++){
            a[i][j] = a[i][j] - l[i][k] * u_k[j]; // Notice u_k[j]
        }
}
```

Section	Time	Section	Time	Section	Time
max	2	max	2	max	2
swap	9	swap	17	swap	40
lu_upd	7	lu_upd	10	lu_upd	23
a_upd	375	a_upd	221	a_upd	224
Total	397	Total	254	Total	293
t	**2**	t	**4**	t	**8**

Lessons learnt: This optimization gave us the first hint that hoisting of repetitive computations can be beneficial for the program runtime.

Iteration - II

Intuition :

The "a_upd" phase consists of the following loop

```
// Outer Loop
for(int i = k+1; i < n; i++){
        double* u_k = u[k];
        // Inner Loop
        for(int j = k+1; j < n; j++){
            a[i][j] = a[i][j] - l[i][k] * u_k[j]; // Notice l[i][k]
        }
}
```

Observe how the value of $l[i][k]$ remains constant throughout the execution of the inner loop as well! Instead of making repeated memory access calls, we can set this parameter to constant.

```
// Outer Loop
for(int i = k+1; i < n; i++){
        double* u_k = u[k];
        double l_i_k = l[i][k];                   // Notice l_i_k
        // Inner Loop
        for(int j = k+1; j < n; j++){
            a[i][j] = a[i][j] - l_i_k * u_k[j]; // Notice l_i_k
        }
}
```

Section	Time	Section	Time	Section	Time
max	2	max	2	max	3
swap	10	swap	17	swap	39
lu_upd	7	lu_upd	10	lu_upd	22
a_upd	273	a_upd	164	a_upd	181
Total	296	Total	196	Total	249
t	**2**	t	**4**	t	**8**

Lessons learnt: Hoisting array base address calculations which are loop invariants also helps decrease the runtime.

Iteration - III

Intuition :

```

// Inner Loop
for(int j = k+1; j < n; j++){
    a[i][j] = a[i][j] - l[i][k] * u_k[j]; // Notice a[i][j]
}
```

In the inner loop, notice how $a[i][j]$ is computed for every iteration , even when we know that $a[i]$ occurs contiguously and thus $a[i][j+1]$ begins exactly 8 bytes after $a[i][j]$! A similar argument applies for $u\_k[j]$ as well.

After using pointer increments instead of accesses; storing pointers, and incrementing them, in order to do optimised pointer calculation rather than adding j*8 to base index for every j. :

```
a_exp = &(a[i][k]);                // Notice pointer for a and u access
for(int j = k; j < n; j++){
    *(a_exp) -= l_ik * (*u_exp);
    a_exp++; u_exp++;
}
```

Section	Time	Section	Time	Section	Time
max	2	max	2	max	2
swap	9	swap	17	swap	40
lu_upd	7	lu_upd	10	lu_upd	23
a_upd	375	a_upd	221	a_upd	224
Total	397	Total	254	Total	293
t	**2**	t	**4**	t	**8**

Lessons learnt: Changing the array accesses to pointer operations does not provide an advantage, probably due to the compiler easily being able to recognize array accesses and perhaps optimize accordingly.

Iteration - IV

Intuition :

Splitting work using *#pragma omp parallel* instead of *#pragma omp parallel for* to gain more fine gain control, and using sequential code if k is too low to avoid paying unnecessary overheads.

Section	Time	Section	Time	Section	Time
max	1	max	2	max	2
swap	9	swap	17	swap	37
lu_upd	7	lu_upd	11	lu_upd	21
a_upd	240	a_upd	147	a_upd	153
Total	261	Total	182	Total	217
t	**2**	t	**4**	t	**8**

We also tried experimenting with *#pragma omp parallel for schedule(dynamic)* i.e. dynamic scheduling of loops in OpenMP, but this did not have a significant impact.

Lessons learnt : Finer grained control indeed provides an advantage for parallel programs. Unequal distribution of work seems like a small problem, but perhaps the cost adds up over iterations and increases the runtime significantly.

Iteration - V

Intuition : We wanted to experiment with a different data layout for our matrices - column major instead of row major. We decided to lay out A and L as column major, and U as row major. The considerations were the following:

- The max-finding step would be more efficient, as we now need to find the maximum element of a row instead of a column. It would also be amenable to parallelization.
- The swaps would be much more inefficient. For the row-major case, for swapping rows of A we only needed to swap two pointers, and for updating L we used the efficient *memcpy*, whereas for a column major layout we need to swap elements individually.
- The LU update stage is more or less indifferent.

  ```
  for i = k+1 to n
      l(k,i) = a(k,i)/u(k,k) //S1
      u(k,i) = a(i,k) //S2
  ```

  Now, the accesses to L and A are optimized in S1, however, the access to A is also suboptimal in S2 (the loop is now iterating over the 1st dimension).
- In the A update stage we perform a loop interchange, which makes the psuedocode:

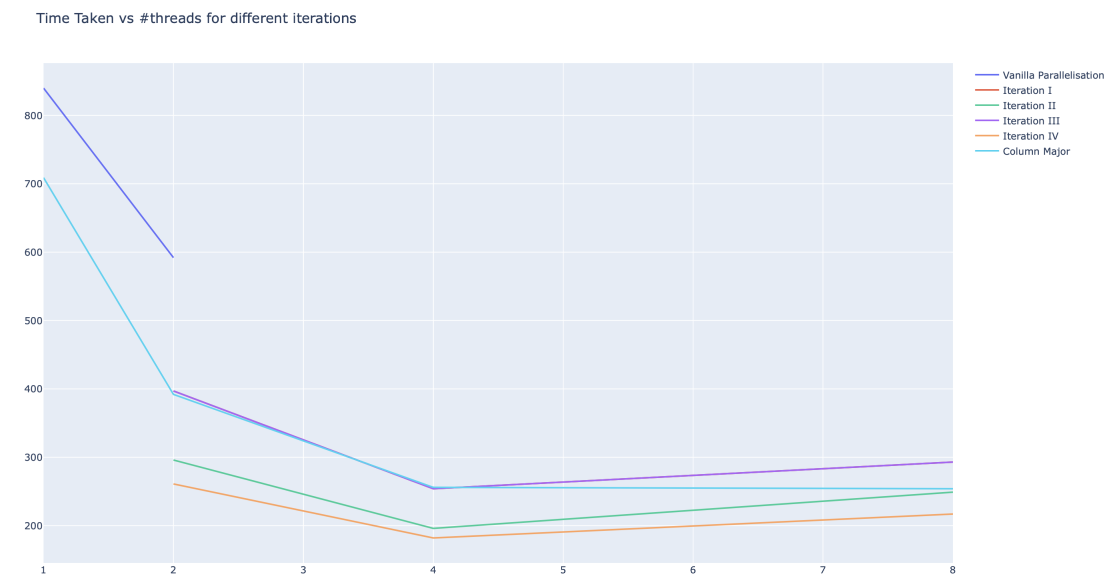
  ```
  for i = k+1 to n
      for j = k+1 to n
        a(i,j) = a(i,j) - l(k,j)*u(k,i)
  ```

  By switching the loop, the access to A and U has been made optimal. The access of L is also cache-friendly, as the last dimension is accessed by the innermost loop.

Section	Time	Section	Time	Section	Time	Section	Time
Total	709	Total	392	Total	256	Total	254
t	**1**	t	**2**	t	**4**	t	**8**

Lessons learnt : Unfortunately, the increase in the time for the swapping dominated any decrease in the A update step (which was quite small), and the overall performance of the code was more or less the same, if not slightly worse.

Final Optimisations



Above we have plotted the runtimes for the different iterations of our implementation. The vanilla implementation has been plotted for small values of threads ($t = 1$ and $t = 2$), the column major implementation has been plotted for $t = 1$ to 8, and the rest of the iterations have been plotted from $t = 2$ to 8. As we can see from the graph, Iteration IV (in orange) has the most efficient implementation for OpenMP.

pthreads

For the pthread part of the assignment, we started with our optimized OpenMP implementation as a reference point. The pthread library does not provide most of the high level functionalities provided by OpenMP, due to which we were required to make functions as a target for launching the threads, and create global variables for the shared data to pass to the threads.

First Iteration

For a first iteration, we simply transformed our OpenMP code to explicitly launch threads using the `pthread_create()` and `pthread_join()` functionalities. Since we also had to manually divide the work up amongst the threads statically, we tried to ensure that all threads do an equal amount of work.

Section	Time	Section	Time	Section	Time
Total	261	Total	182	Total	217
t	**2**	t	**4**	t	**8**

We notice that the overall time taken by pthread is significantly higher than that taken by OpenMP. On a closer inspection of the breakup of the time taken, we see that the time taken for individual components also follows this trend.

For example, the function to calculate the max takes 1-2 ms for the OpenMP program, but takes 40-50ms for the pthread implementation. This is likely due to the overhead of thread launching. We confirmed this by launching threads that target an empty function which just returns. Even this computation takes 30-40ms to execute.

Second Iteration

Since we realized that the repeated forking and joining to create threads was creating a large overhead, we decided to alter the structure of our pthread program. In our modified program, we launched t threads at the beginning of the program. Instead of joining the threads, we use a pthread barrier, and call the `pthread_barrier_wait()` function to act as a fence whenever we transition from parallel to sequential code. Further, we only let thread 0 run the sequential code.
We conjectured that this should avoid the additional overhead of fork-join whenever we spawn new threads, especially for matrices of smaller sizes (since the difference was visible for $n = 1000$).

Section	Time	Section	Time	Section	Time
Total	261	Total	182	Total	217
t	**2**	t	**4**	t	**8**

Here, we observe an improvement in the overall runtime of our program for $n = 1000$ sized matrices.

Final Results

When tested for times on $S1$ for an n = 8000 matrix, the sequential code ran in 610.086 seconds, and the following times were obtained for the parallelised codes. Note that all times are in seconds .

Threads	$Time_{pthread}$	$S_{pthread}$	$\epsilon_{pthread}$	$Time_{OpenMP}$	S_{OpenMP}	ϵ_{openMP}
1	704.558	1	1	450.248	1	1
2	417.096	1.613	0.807	249.121	1.807	0.904
4	239.217	2.856	0.714	126.455	3.561	0.890
8	161.268	4.233	0.529	76.328	5.899	0.737
16	147.128	4.672	0.292	70.074	6.425	0.402

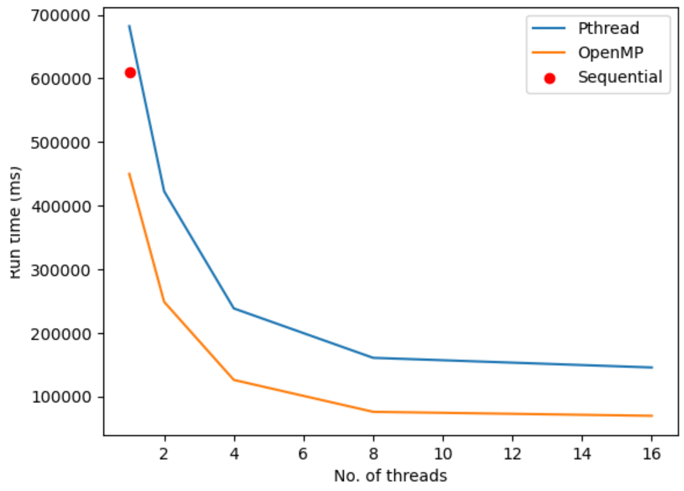


Figure 1: Plot of Runtime (in ms) v/s No of Threads (for both the implementations)

Final Implementation Details

The following are the final implementation details for the pthread and OpenMP programs:

- The matrices A, L, U are maintained as an array of pointers to arrays (of doubles), with each array of doubles being allocated using `malloc` separately. We keep this design to prevent false sharing. In our implementation, we ensure that we divide work such that each thread gets to work on a (set of) rows of the matrices. This data layout makes sure that no two threads work on data in the same cache line.
- We do not parallelize the max-finding step as there is no speedup, but rather a parallelization overhead. For the swaps, we use a pointer exchange and the *memcpy* function which turns out to be faster than parallellizing manually.

- We parallelize the loops which compute the LU updates and the A updates. We explicitly ensure that the work is divided equally.
- We hoist redundant computations in loops outside, such as the repeated computation of $U[k][k]$, the array base address computation $L[k]$, etc.
- For pthreads, we use the concept of keeping a thread pool initially, which is discussed in the second iteration of the pthread section.
- We did not use the `-O3` flag for optimizing the parallel implementation in the final submission, although it increases the speedup, because it would be unfair to compare compiler optimized code with the unoptimized sequential code.
- We allocate the matrices dynamically using `malloc`, as declaring the arrays on the stack was causing the program to crash.

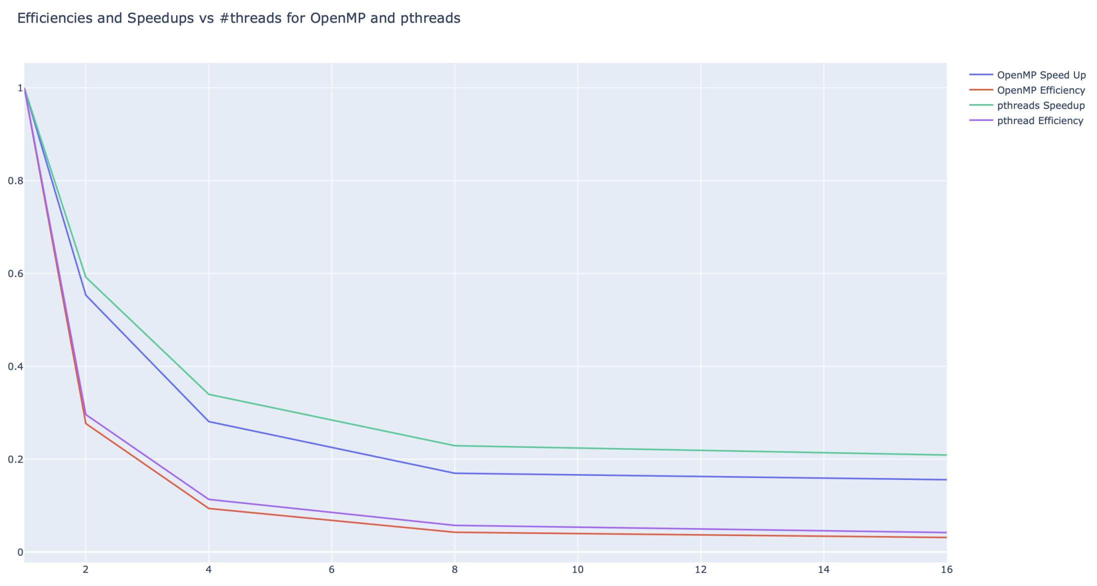


Figure 2: Plot of Efficiency v/s No of Threads (for both the implementations)