MULTITHREADED LU DECOMPOSITION

ASSIGNMENT-1: COL380 – INTRO TO PARALLEL AND DISTRIBUTED PROGRAMMING

Nisarg Bhatt – 2017CS10354
Divyanshu Mandowara – 2017CS10333

HOW DOES THE PROGRAM WORK?

- We analyze the sequential code and first identify the regions where we can exploit parallelism. The outer "for" loop in the program cannot be parallelized as there are data dependencies (the ith step needs to be done before the (i+1)th step and so on). See figure-1.
- Inside the outer for loop, there are other linear loops and one quadratic loops (the double-for loop).
- Functionality of the double-for loop is to update each element of the residual matrix. This can be done independently for each element in it and hence, can be parallelized.
- Also, row swapping operations can be parallelized as a swap in a given index doesn't depend on other indices.

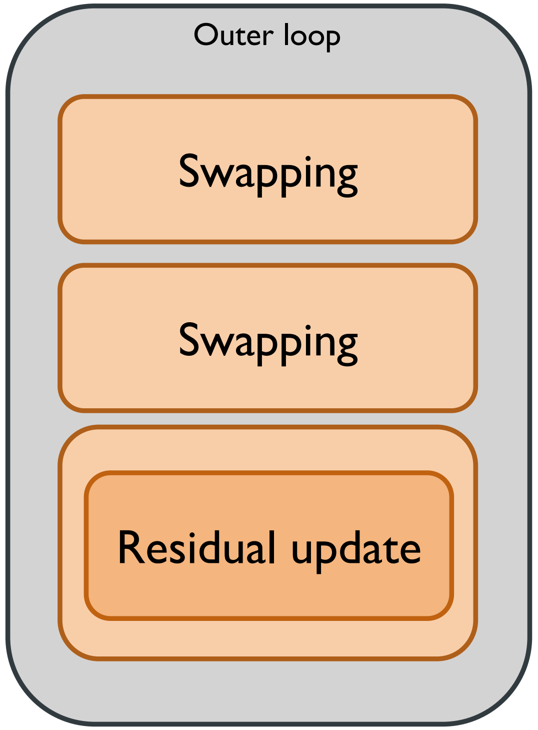
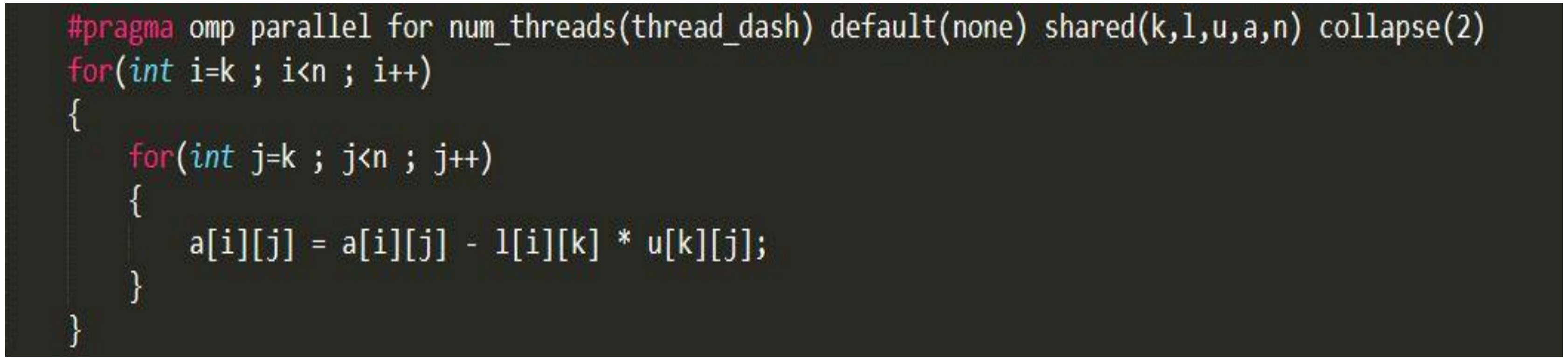


Figure-1

OPENMP IMPLEMENTATIONS

Double For loop (we used `#pragma omp parallel for` **`collapse(2`**`)` which is basically used to give directions to OpenMP to parallelize nested loops). [Fig.-2]



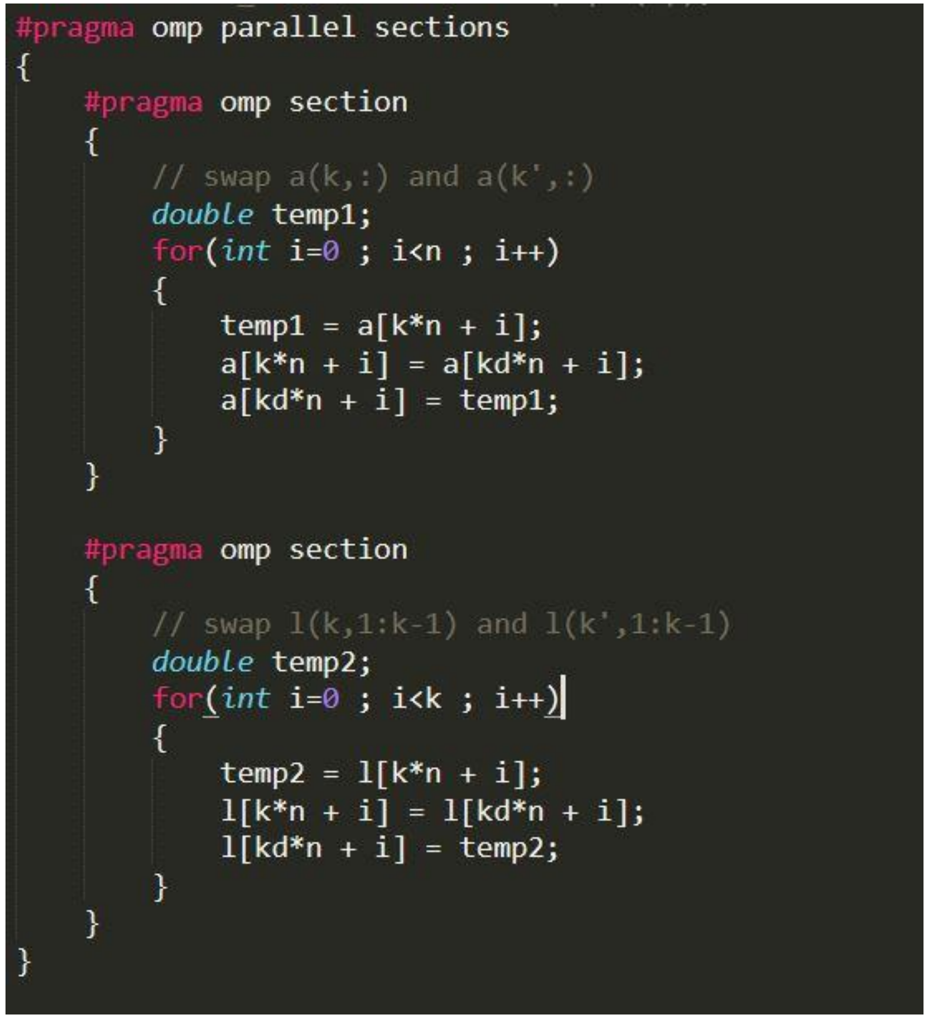
```
#pragma omp parallel for num_threads(thread_dash) default(none) shared(k,l,u,a,n) collapse(2)
for(int i=k ; i<n ; i++)
{
    for(int j=k ; j<n ; j++)
    {
        a[i][j] = a[i][j] - l[i][k] * u[k][j];
    }
}
```

Figure-2

OPENMP IMPLEMENTATIONS

`#pragma omp parallel sections`

used to tell OpenMP to execute swap_a and swap_l operations together. [Fig.-3]

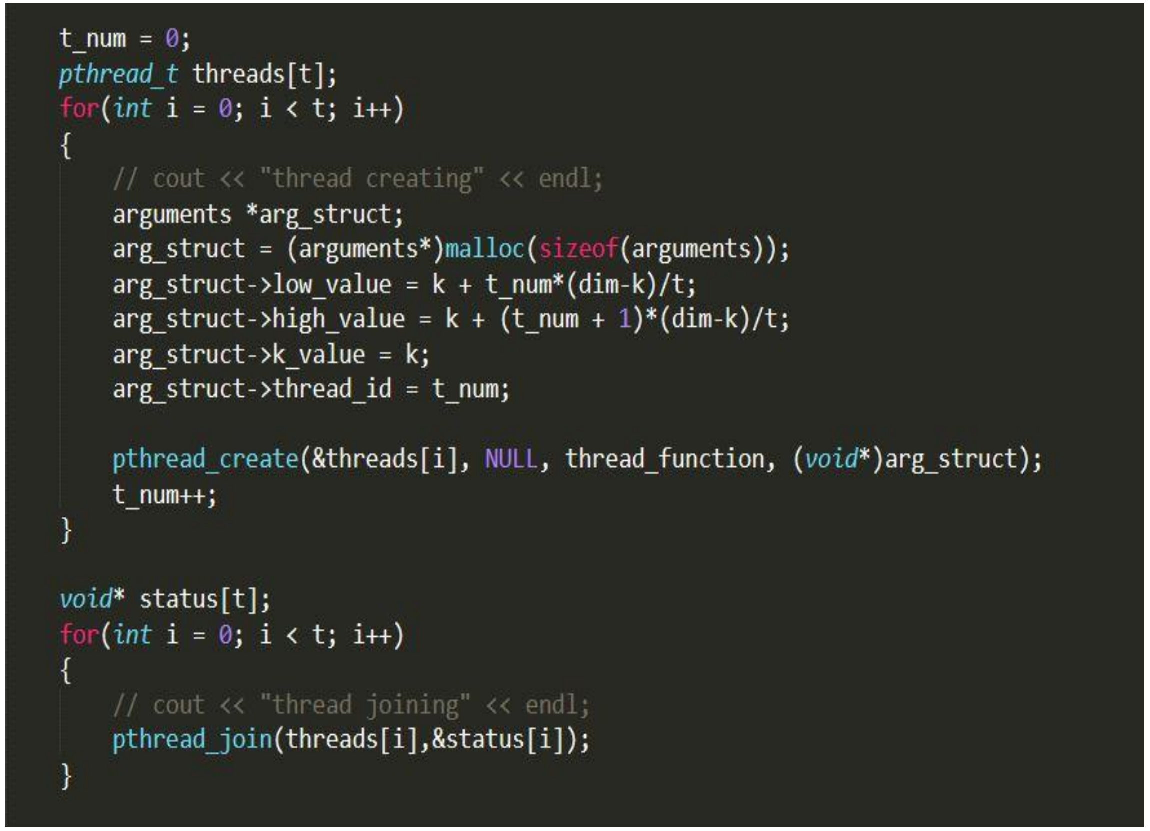


```
#pragma omp parallel sections
{
    #pragma omp section
    {
        // swap a(k,:) and a(k',:)
        double temp1;
        for(int i=0 ; i<n ; i++)
        {
            temp1 = a[k*n + i];
            a[k*n + i] = a[kd*n + i];
            a[kd*n + i] = temp1;
        }
    }

    #pragma omp section
    {
        // swap l(k,1:k-1) and l(k',1:k-1)
        double temp2;
        for(int i=0 ; i<k ; i++)
        {
            temp2 = l[k*n + i];
            l[k*n + i] = l[kd*n + i];
            l[kd*n + i] = temp2;
        }
    }
}
```

Figure-3

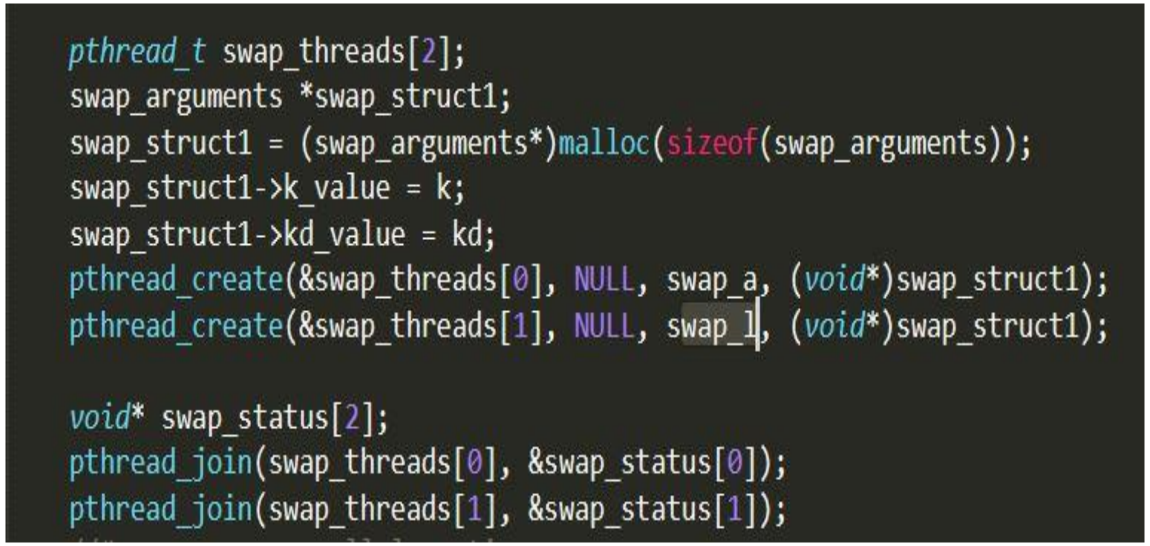
PTHREAD IMPLEMENTATIONS

Figure-4: Pthread implementation for parallelizing double for loop (create t threads; divide work; call thread_function; synchronize)



```
t_num = 0;
pthread_t threads[t];
for(int i = 0; i < t; i++)
{
    // cout << "thread creating" << endl;
    arguments *arg_struct;
    arg_struct = (arguments*)malloc(sizeof(arguments));
    arg_struct->low_value = k + t_num*(dim-k)/t;
    arg_struct->high_value = k + (t_num + 1)*(dim-k)/t;
    arg_struct->k_value = k;
    arg_struct->thread_id = t_num;

    pthread_create(&threads[i], NULL, thread_function, (void*)arg_struct);
    t_num++;
}

void* status[t];
for(int i = 0; i < t; i++)
{
    // cout << "thread joining" << endl;
    pthread_join(threads[i],&status[i]);
}
```

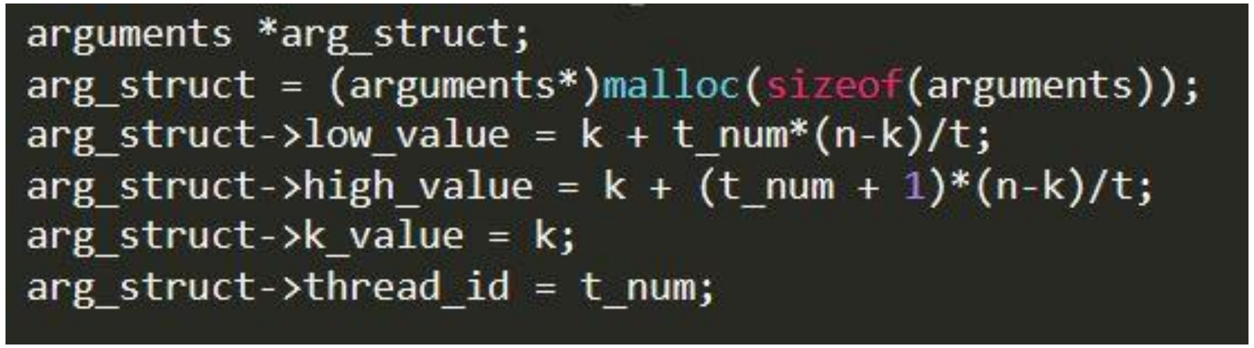
Figure-5: Pthread implementation for parallelizing the swap_a and swap_l (create 2 threads; divide work; call swap_a and swap_l; join threads)



```
pthread_t swap_threads[2];
swap_arguments *swap_struct1;
swap_struct1 = (swap_arguments*)malloc(sizeof(swap_arguments));
swap_struct1->k_value = k;
swap_struct1->kd_value = kd;
pthread_create(&swap_threads[0], NULL, swap_a, (void*)swap_struct1);
pthread_create(&swap_threads[1], NULL, swap_l, (void*)swap_struct1);

void* swap_status[2];
pthread_join(swap_threads[0], &swap_status[0]);
pthread_join(swap_threads[1], &swap_status[1]);
```

EXPLOITING PARALLELISM AND PARTITIONING WORK (PTHREADS)

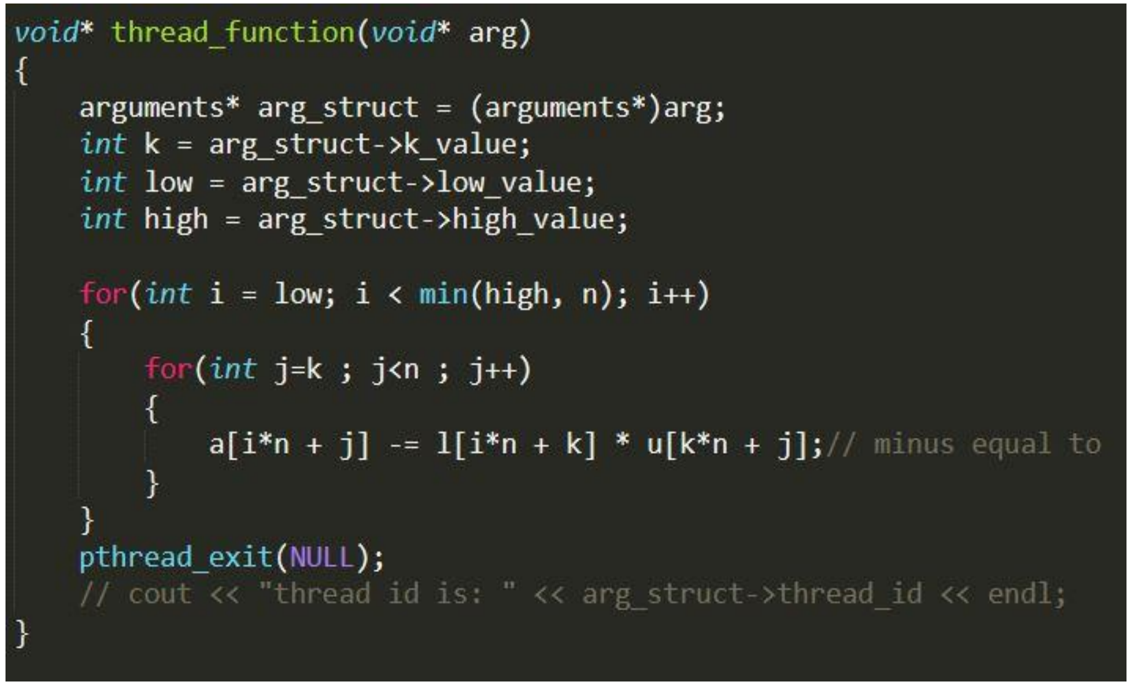
- Every thread works on the thread_function starting from row index "low" to row index "high". That is to say, every thread works on (high – low) number of rows and does the "a" matrix update.
- The low and high indices are assigned uniformly to all the threads in a static schedule with chunk size (n - k)/t where n is the dimension of matrix, k is the index of outer for loop and t is the number of the threads.
- No two threads are updating the same variable so all can work in parallel.



```
arguments *arg_struct;
arg_struct = (arguments*)malloc(sizeof(arguments));
arg_struct->low_value = k + t_num*(n-k)/t;
arg_struct->high_value = k + (t_num + 1)*(n-k)/t;
arg_struct->k_value = k;
arg_struct->thread_id = t_num;
```

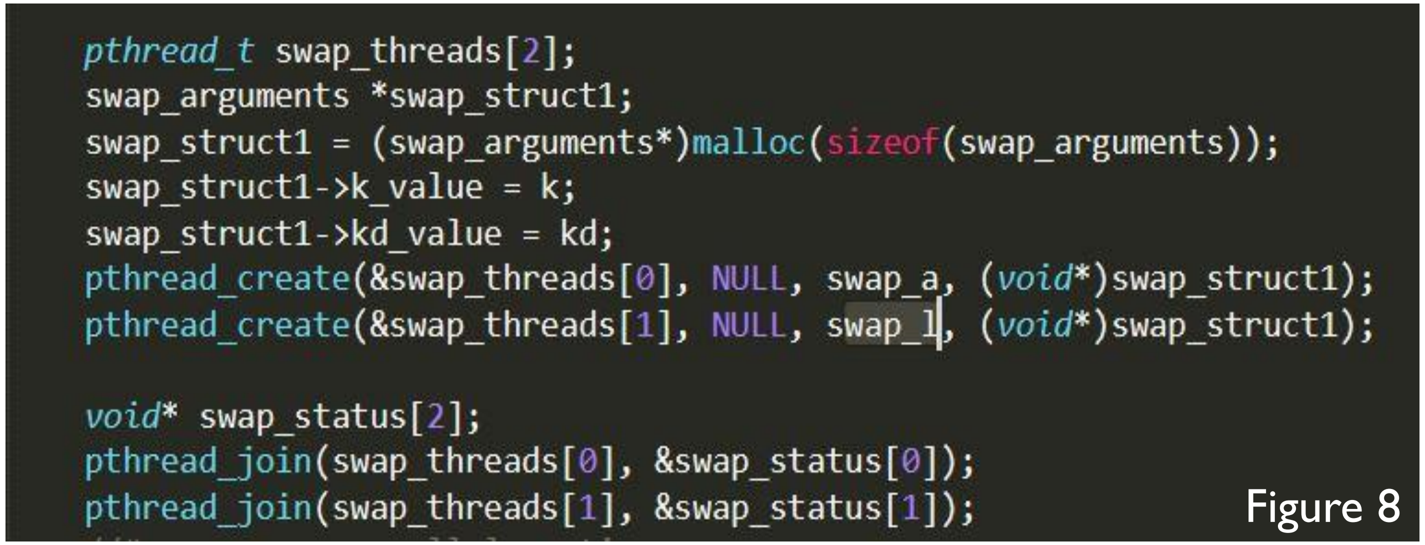
Figure-6: Argument struct for every thread

Figure-7: thread_function



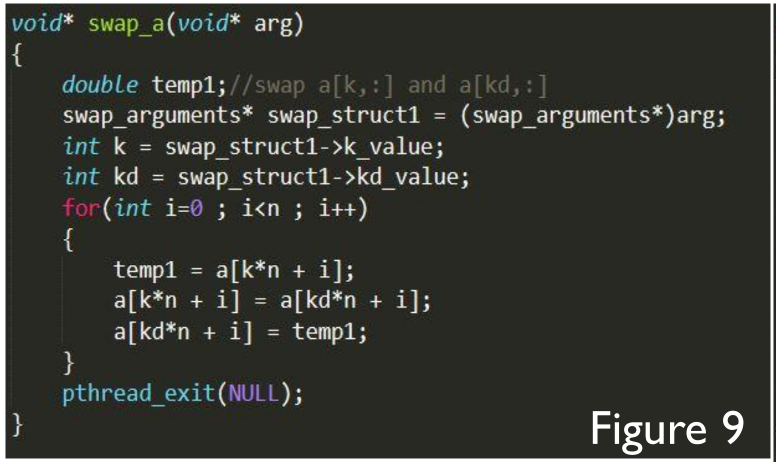
```
void* thread_function(void* arg)
{
    arguments* arg_struct = (arguments*)arg;
    int k = arg_struct->k_value;
    int low = arg_struct->low_value;
    int high = arg_struct->high_value;

    for(int i = low; i < min(high, n); i++)
    {
        for(int j=k ; j<n ; j++)
        {
            a[i*n + j] -= l[i*n + k] * u[k*n + j];// minus equal to
        }
    }
    pthread_exit(NULL);
    // cout << "thread id is: " << arg_struct->thread_id << endl;
}
```

EXPLOITING PARALLELISM AND PARTITIONING WORK (PTHREADS)



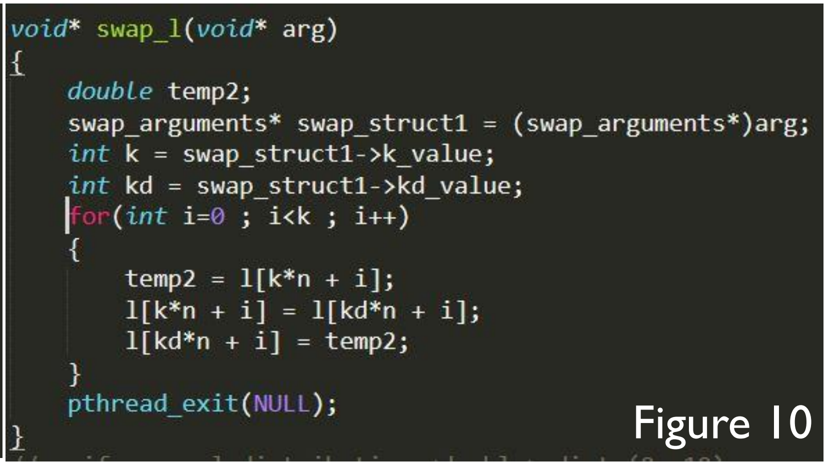
```
pthread_t swap_threads[2];
swap_arguments *swap_struct1;
swap_struct1 = (swap_arguments*)malloc(sizeof(swap_arguments));
swap_struct1->k_value = k;
swap_struct1->kd_value = kd;
pthread_create(&swap_threads[0], NULL, swap_a, (void*)swap_struct1);
pthread_create(&swap_threads[1], NULL, swap_l, (void*)swap_struct1);

void* swap_status[2];
pthread_join(swap_threads[0], &swap_status[0]);
pthread_join(swap_threads[1], &swap_status[1]);
```

Figure 8



```
void* swap_a(void* arg)
{
    double temp1;//swap a[k,:] and a[kd,:]
    swap_arguments* swap_struct1 = (swap_arguments*)arg;
    int k = swap_struct1->k_value;
    int kd = swap_struct1->kd_value;
    for(int i=0 ; i<n ; i++)
    {
        temp1 = a[k*n + i];
        a[k*n + i] = a[kd*n + i];
        a[kd*n + i] = temp1;
    }
    pthread_exit(NULL);
}
```

Figure 9



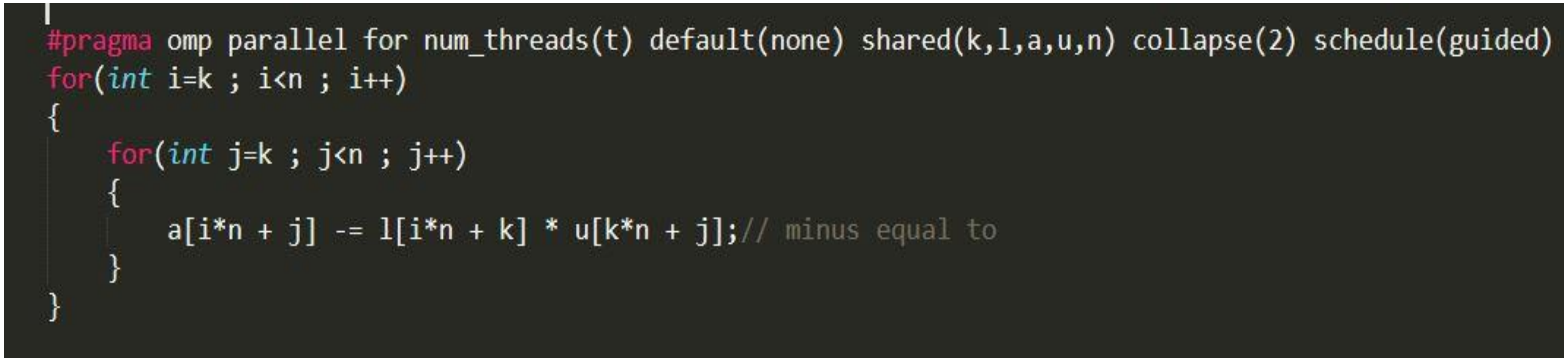
```
void* swap_l(void* arg)
{
    double temp2;
    swap_arguments* swap_struct1 = (swap_arguments*)arg;
    int k = swap_struct1->k_value;
    int kd = swap_struct1->kd_value;
    for(int i=0 ; i<k ; i++)
    {
        temp2 = l[k*n + i];
        l[k*n + i] = l[kd*n + i];
        l[kd*n + i] = temp2;
    }
    pthread_exit(NULL);
}
```

Figure 10

- Swap_a and Swap_l operations are independent.
- They can be executed in parallel.
- So, we created two threads and gave one thread to swap_a and one to swap_l.
- Finally call pthread_join() to wait for both threads to finish their tasks.

EXPLOITING PARALLELISM AND PARTITIONING WORK (OPENMP)

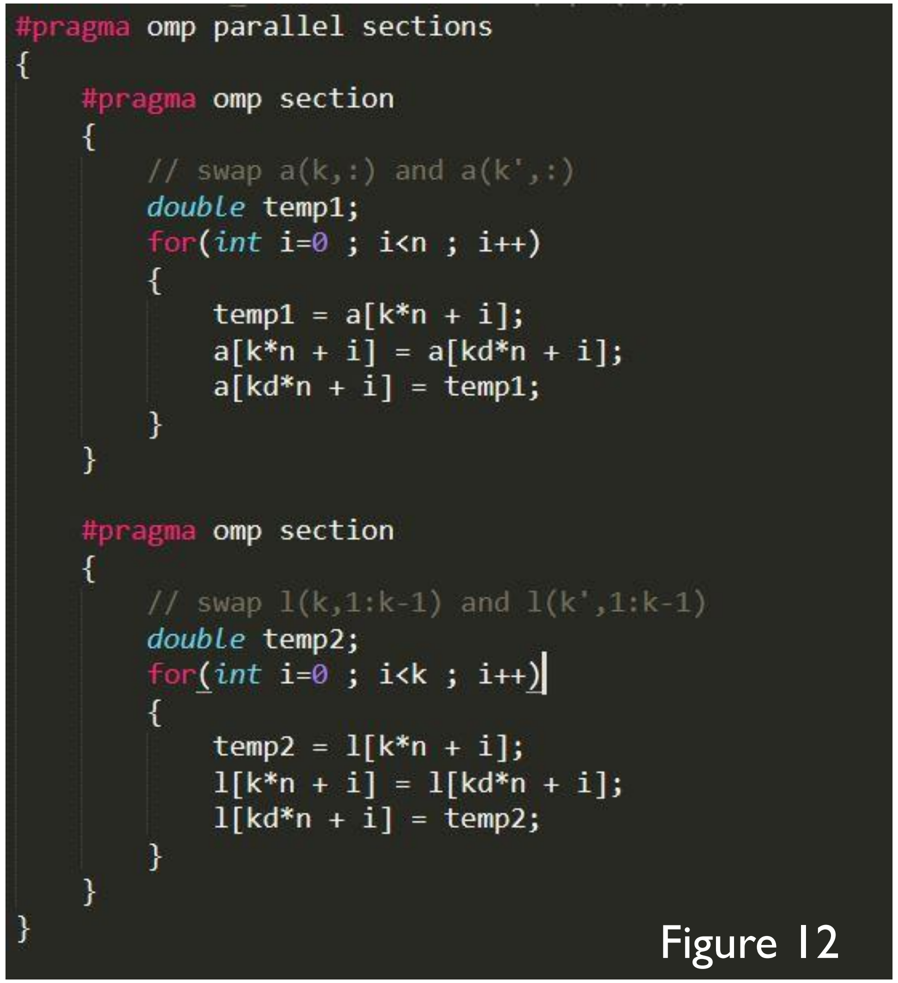
- We used "**collapse(2)**" pragma in openmp to guide openmp to divide the work in the nested loops.
- We tried for different schedules for partitioning the work and got the best results from the **"guided"** schedule.
- In a guided schedule, the size of a chunk is proportional to the number of unassigned iterations divided by the number of the threads. Therefore the size of the chunks decreases as the remaining work decreases.



```
#pragma omp parallel for num_threads(t) default(none) shared(k,l,a,u,n) collapse(2) schedule(guided)
for(int i=k ; i<n ; i++)
{
    for(int j=k ; j<n ; j++)
    {
        a[i*n + j] -= l[i*n + k] * u[k*n + j];// minus equal to
    }
}
```

Figure 11

EXPLOITING PARALLELISM AND PARTITIONING WORK (OPENMP)



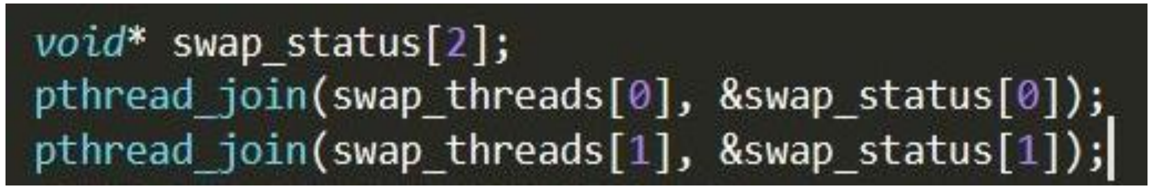
```
#pragma omp parallel sections
{
    #pragma omp section
    {
        // swap a(k,:) and a(k',:)
        double temp1;
        for(int i=0 ; i<n ; i++)
        {
            temp1 = a[k*n + i];
            a[k*n + i] = a[kd*n + i];
            a[kd*n + i] = temp1;
        }
    }

    #pragma omp section
    {
        // swap l(k,1:k-1) and l(k',1:k-1)
        double temp2;
        for(int i=0 ; i<k ; i++)
        {
            temp2 = l[k*n + i];
            l[k*n + i] = l[kd*n + i];
            l[kd*n + i] = temp2;
        }
    }
}
```

Figure 12

- `#pragma omp parallel sections`
- The first section contains the code of swapping row of matrix “a”.
- The second section contains the code of swapping the row of matrix “l”.
- This pragma tells openmp to distribute this work among two threads and execute them in parallel.

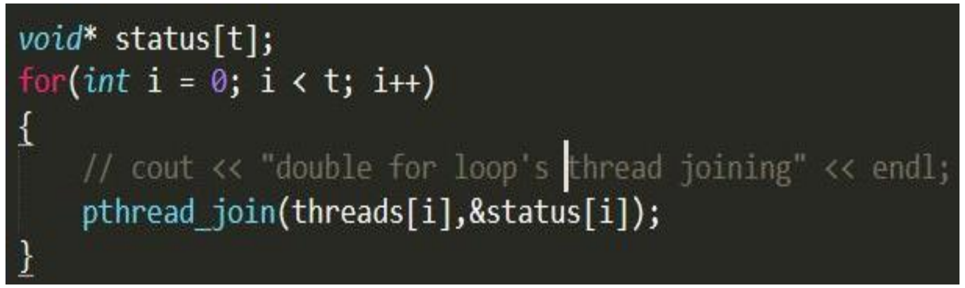
SYNCHRONIZING

- OpenMP has an implicit barrier at the end of the for loop and sections so need to do any other thing for synchronizing.
- For Pthreads:
  - After every thread does the work assigned to it, it calls pthread_exit();
  - At the end, we call pthread_join() to collect the status of every thread and proceed further in the algorithm only after all threads have finished their tasks.
  - The target matrix is updated and the algorithm resumes.



```
void* swap_status[2];
pthread_join(swap_threads[0], &swap_status[0]);
pthread_join(swap_threads[1], &swap_status[1]);
```

Figure13: Swap_a and swap_l threads



```
void* status[t];
for(int i = 0; i < t; i++)
{
    // cout << "double for loop's thread joining" << endl;
    pthread_join(threads[i],&status[i]);
}
```

Figure 14: Double for loop’s threads

VARIOUS IMPLEMENTATIONS

- Apart from OpenMP and Pthreads, we also had multiple other implementations.
- We used two ways to allocate the data structures to the matrices "a", "l" and "u".
  - Array of n pointers to n-element data arrays. (**SPARSE**)
  - A contiguous array of n*n elements. (**DENSE**)
- We used two parallelization strategies after doing several experiments:
  - Parallelizing only the double for loop equally dividing the work among threads. (Double For)
  - Along with parallelizing the double for loop, also carry the swap_a and swap_l functions in parallel. (Double for with Parallel sections)

Going from Double For to Double For with Parallel Sections:

- In order to exploit the fact that swap_a and swap_l have no data dependency, we parallelized both the parts.
- But, it turned out that the overhead due to the thread creation and synchronization resulted in the degrading of performance.

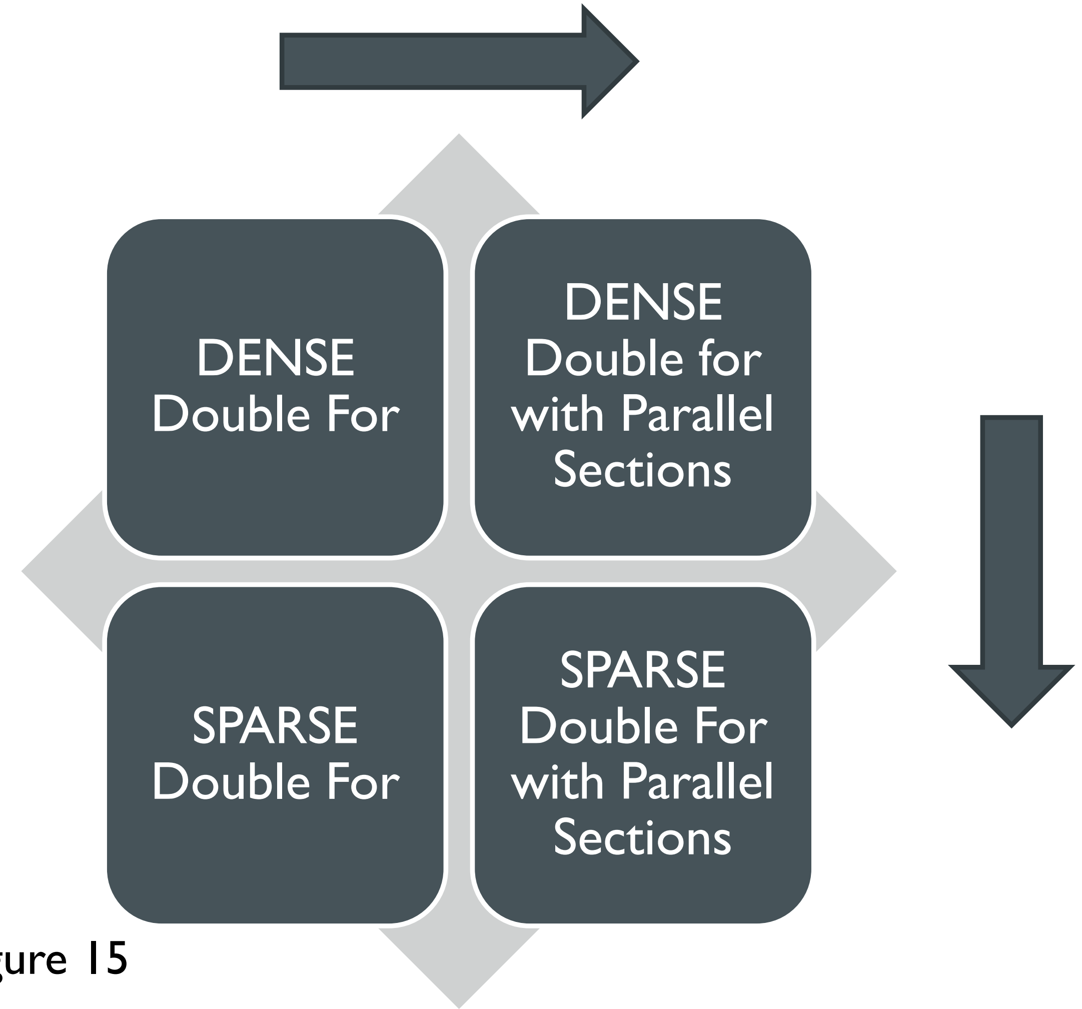

DENSE Double For
DENSE Double for with Parallel Sections
SPARSE Double For
SPARSE Double For with Parallel Sections

Figure 15

Going from DENSE to SPARSE:

- Using shared data structures is a potential source of **false sharing** so we changed the matrices a, l and u from contiguous array of n*n to an array of n pointers to n-element data arrays and then divided the work among threads so that they don't try to access adjacent memory locations.
- This helped in improving the performance.

TIMING MEASUREMENTS

- OPENMP implementation on a matrix size of 7000 (time in seconds; T denote thread count)

Implementation	Sequential	T = 1	T = 2	T = 4	T = 6	T = 8	T = 12	T = 16	T = 32
SPARSE Double for	408.7	414.15	221.29	128.46	104.89	116.44	103.23	97.92	95.44
SPARSE with parallel sections		440.24	283.25	150.38	148.03	150.23	110.61	138.54	169.4
DENSE Double for	384.58	386.01	198.89	116.03	94.15	109.77	99.85	98.28	96.6
DENSE with parallel sections		434.08	272.21	154.85	142.08	140.81	98.69	127.88	167.39

Table 1

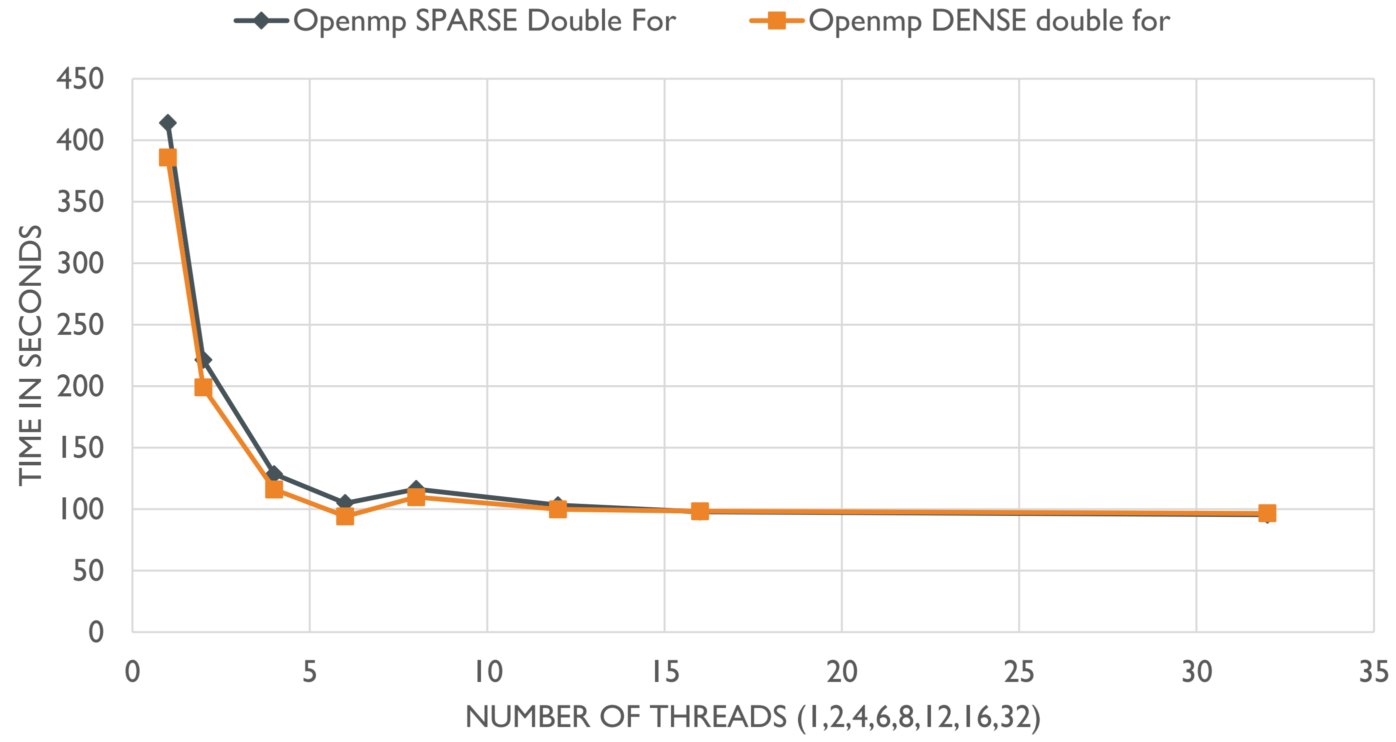
TIMING MEASUREMENTS

- PTHREADS implementation on a matrix size of 7000 (time in seconds; T denotes thread count)

Implementation	Sequential	T = 1	T = 2	T = 4	T = 6	T = 8	T = 12	T = 16	T = 32
SPARSE Double for	408.7	397.74	204.91	113.69	106.46	113.23	104.42	102.29	98.43
SPARSE with parallel sections		414.54	215.87	120.4	105.62	115.95	102.75	109.35	98.08
DENSE Double for	384.58	412.63	217.04	126.83	111.45	111.3	103.9	105.5	100.07
DENSE with parallel sections		414.73	211.8	124.41	106.12	114.16	98.63	102.27	96.69

Table 2

OPENMP (TIME VS NUMBER OF THREADS)



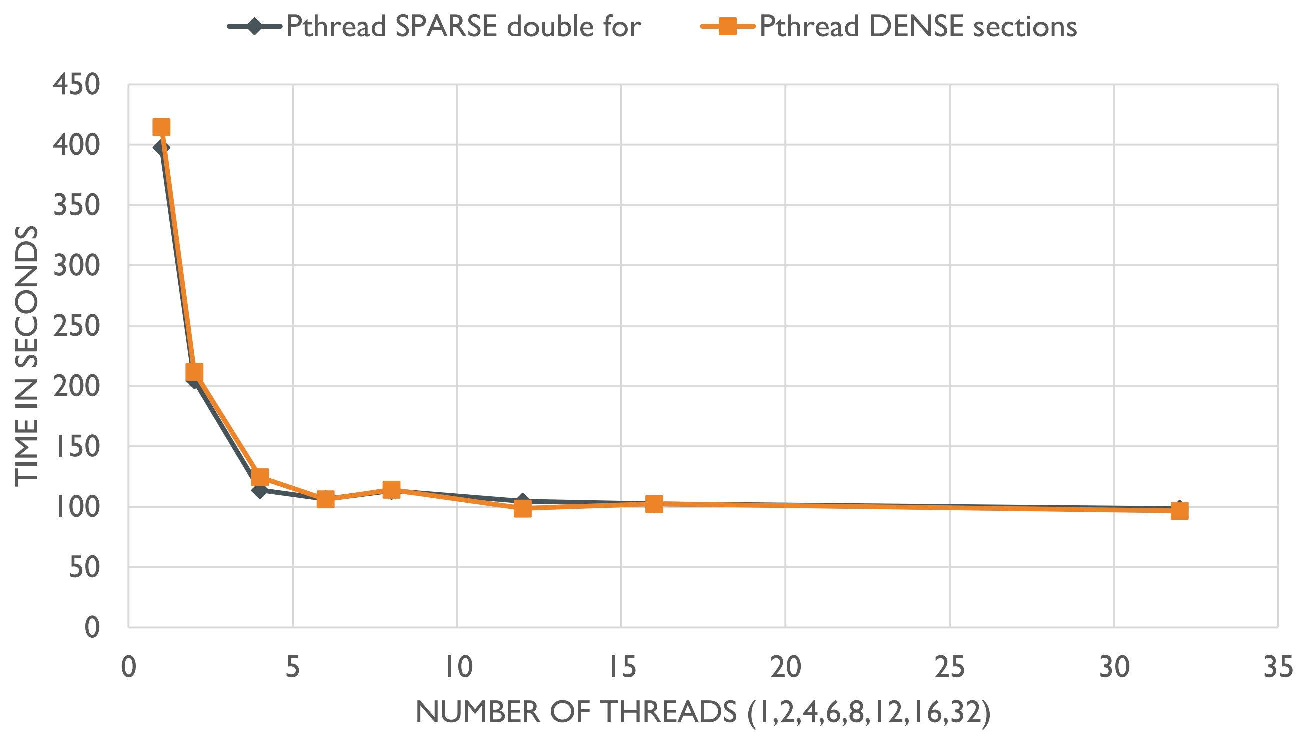
Graph 1

TIME MEASUREMENTS

OPENMP DENSE vs OPENMP SPARSE

(both for “Double For” only as it was better than the parallel sections”)

PTHREAD (TIME VS NUMBER OF THREADS)



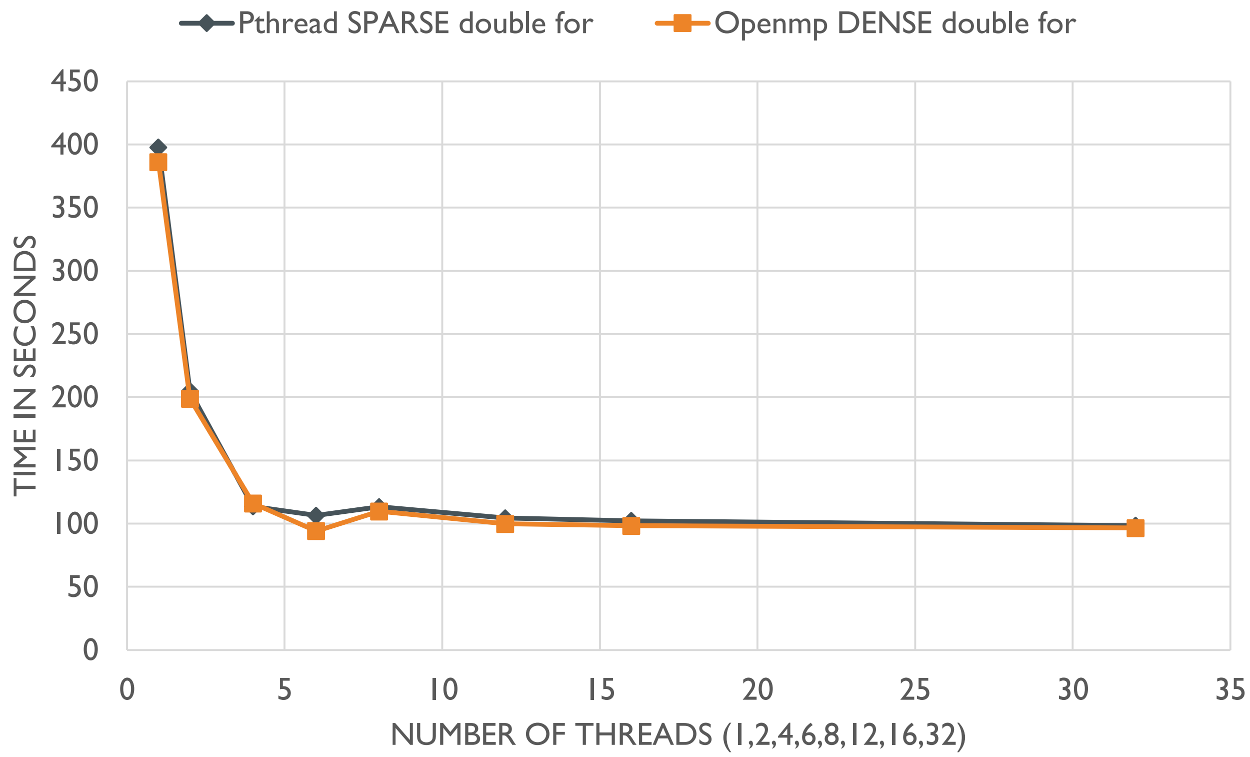
Graph 2

TIME MEASUREMENTS

PTHREADS DENSE vs PTHREADS SPARSE

(“Double For” for SPARSE and “Parallel section” for DENSE)

PTHREAD VS OPENMP (TIME COMPARISON)

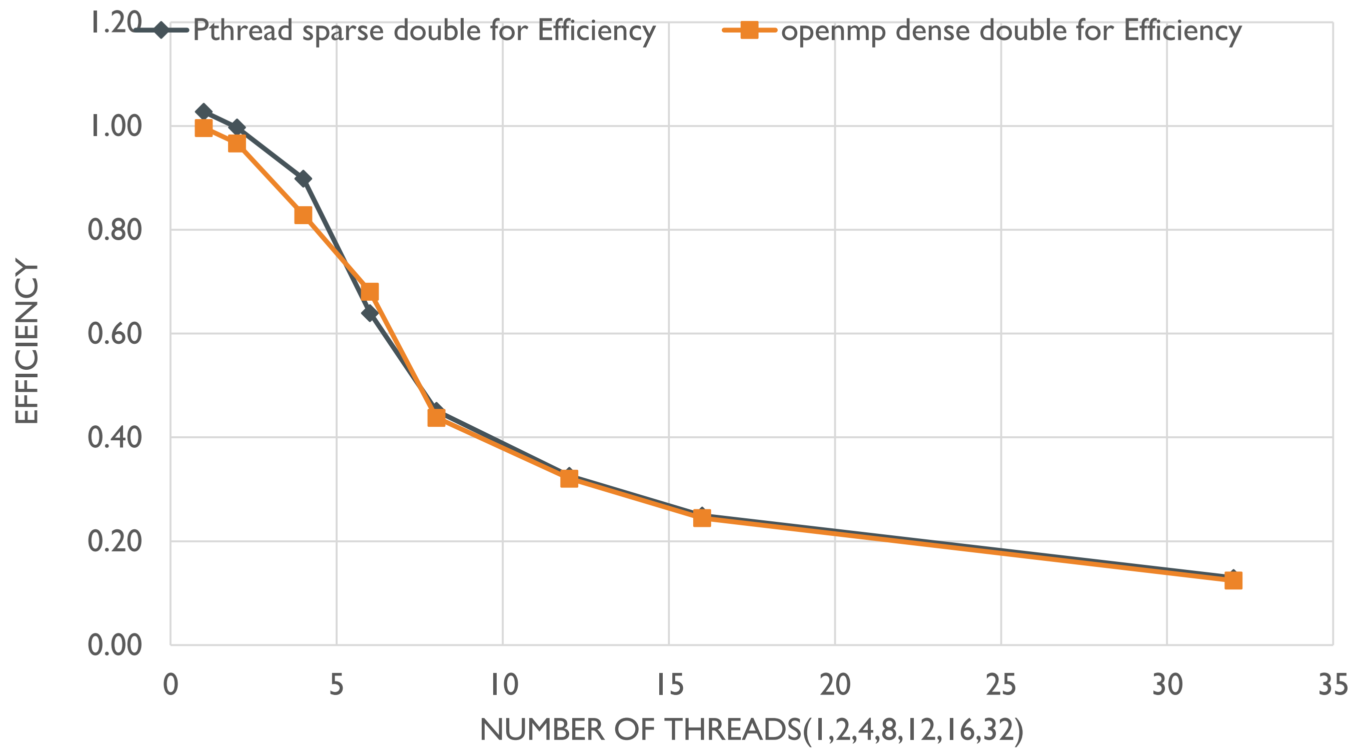


Graph 3

TIME MEASUREMENTS

PTHREADS vs OPENMP

EFFICIENCY COMPARISON



Graph 4

PARALLEL EFFICIENCY

Parallel efficiency is defined as the speedup gained divide by the number of threads.

Pthread vs Openmp

FALSE SHARING

- False sharing occurs when threads on different processors modify variables that reside on the same **cache line**. This invalidates the cache line and forces a memory update to maintain cache coherency.
- Instances where threads access global or dynamically allocated shared data structures are potential sources of false sharing.
- So we changed the matrices "a", "l" and "u" from DENSE (contiguous array) to SPARSE (array of pointers) and then divided the work among threads so that they don't try to access adjacent memory locations.
- Further, in order to minimize the run-time false sharing, we used the compiler optimization flag O3. The compiler eliminates false sharing using thread-private temporal variables.
- Similarly, we minimized the use of global variables by threads and used thread-private variables.

MACHINE SPECIFICATIONS

- !6 GB DDR4 RAM
- 6 Cores (12 logical cores)
- 2.2 GHz intel i7-8750H processor
- Ubuntu 18.04 OS

- Link to the complete results: https://docs.google.com/spreadsheets/d/1szKBsG_lJoW_GnECxrQrTBehkbEnBqhPJf8qf14JWeo/edit?usp=sharing