コンピュータ科学特別講義IV 2018 Final Report

Student name: Tran Van Sang
Faculty: Graduate School of ICT, Nakata Laboratory
Student ID: 48-186106

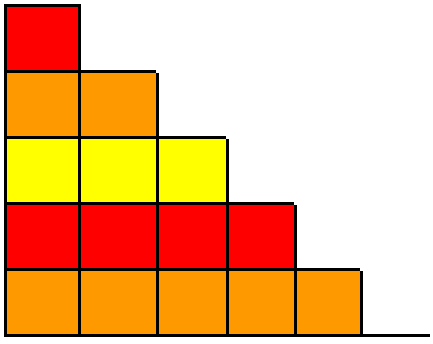
Chosen algorithm: Cholesky linear equation solver serial and parallel version

Source code explanation

- Data generation, and result check is implemented in the program. If the result check fails, program will also fail.
- Serial and parallel algorithms can be runned by switching internal source code flag (both algorithms can be switched to be ON)
  #define RUN_SERIAL
  #define RUN_PARALLEL
- Input size setting can be configured by
  #define N 100
  #define WEAK_SCALING
  If WAK_SCALING is defined, N will be size of input data in each OMP thread. I.e. n = number of threads * number of processes * N when WEAK_SCALING is defined, and n = N otherwise
- The program is based on serial version from Prof. Suda's page at http://sudalab.is.s.u-tokyo.ac.jp/~reiji/PNC18/cholesky_c.txt. There is a bug in his program at line number 92
  ```
  for (i=0; i< n * n; i++)
     xx[i] = 1.0;
  ```
  Should be
  ```
  for (i=0; i< n; i++)
     xx[i] = 1.0;
  ```
- This program uses hybrid (OPM + MPI) model, it requires both MPI and OMP installed. It was tested in Reedbush server.
- Run "make compile" to compile and "make run" to submit job. Edit run.bash to setup number of nodes, processes per node, and number of OMP threads.

How did I parallelize my problem

a. Data distribution



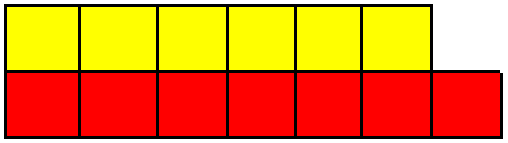


Figure 1. Data distribution

Example with matrix size = 7 x 7 (n = 7), number of processes = 3 (np = 3)
I distributed data one-by-one row to each process. As described in figure 1, row data with same color is distributed into same process. This makes sure in the end of the for loop, data size in all processes keeps being similar.

b. Communication

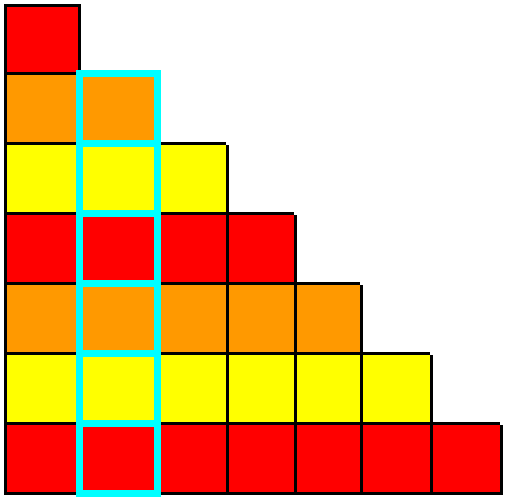


Figure 2: communication between process
To calculate value, each process has to collect information of the respective column from all others. **MPI_allgather** has been used to accomplete this task.
Figure 2 shows example in the second loop, column surrounded by blue color need to be propagated (broadcasted) into all processes. To avoid arranging local data in each time (because broadcasting data does not stay in continuously), local matrices data was **transposed** into column major matrices. Figure 3 illustrates the indexing before and after transpose

c. Parallel computation over processes
In original serial algorithm, we need to execute n-loop to compute sub-matrix of size (n - index). In each computation, we executed another sub-loop for each row of the sub matrix. Total time complexity was $O(n^3)$.
In parallel program, we divided sub-loop of sub matrix computation over multi-process. Each process executes the computation on their own rows.

Process 1	0	1	2	3	4	5	6
	7	8	9	10	11	12	13
	14	15	16	17	18	19	20

Process 2	0	1	2	3	4	5	6
	7	8	9	10	11	12	13
Process 3	0	1	2	3	4	5	6
	7	8	9	10	11	12	13

Process 1	0	3	6	9	12	15	18
	1	4	7	10	13	16	19
	2	5	8	11	14	17	20
Process 2	0	2	4	6	8	10	12
	1	3	5	7	9	11	13
Process 3	0	2	4	6	8	10	12
	1	3	5	7	9	11	13

Figure 3: Data arrangement before and after transpose

d. Parallel computation over OMP threads

OpenMP directive has been injected into most intensive calculation - LDLT calculation

Because the calculation is completely isolated in each sub-loop. If number of OpenMP threads equals to number of physical threads available for each process, ideal strong scale speedup is expected to be linear.

Here is the partial of code that is parallelized by OpenMP

```
    //do LDLT calculation
    //for all rows
    skipped_rows_count = get_nrows(i + 1, np, rank);
    //get_nrows of i + 1 because we are going to skip the row i (do j-loop from i+ 1 to n)
#pragma omp parallel for private(aji, row)
    for(j = skipped_rows_count; j < nrows_local; j++){
      //backup aji
      aji = *ELM(local_a_t, i, j, nrows_local);
      row = get_row(np, rank, j);
      //first elm
      *ELM(local_a_t, i, j, nrows_local) = first_column[row];
      //other elms
#pragma omp parallel for
      for(k = i + 1; k <= row; k++)
        *ELM(local_a_t, k, j, nrows_local) -= aji * first_column[k];
```

```
    }
    MPI(MPI_Barrier(MPI_COMM_WORLD));
  }
```

Benchmark regarding number of processes

a. Weak scaling
Size per process 500
Benchmark data (unit second)
Number of nodes: 1

Number of cores	Running time (seconds)
1	0.1108748913
2	0.4539401531
3	1.044127226
4	1.859787941
5	3.076208115
6	4.663293839
7	6.741144896
8	9.223507881
9	12.93846011
10	18.12153983
11	23.51853299
12	29.34283805
13	38.17646813
14	45.29590106
15	53.64857483
16	62.74412107
17	71.27819204
18	82.06578898
19	91.38895798
20	101.9281061
21	113.7920489
22	126.7263808
23	139.9521151
24	153.6948991
25	169.2541289

26	185.0982232
27	200.1182671
28	218.280077
29	235.0164139
30	252.4360919
31	271.146929
32	290.3599172

Visual graph is shown in figure 4

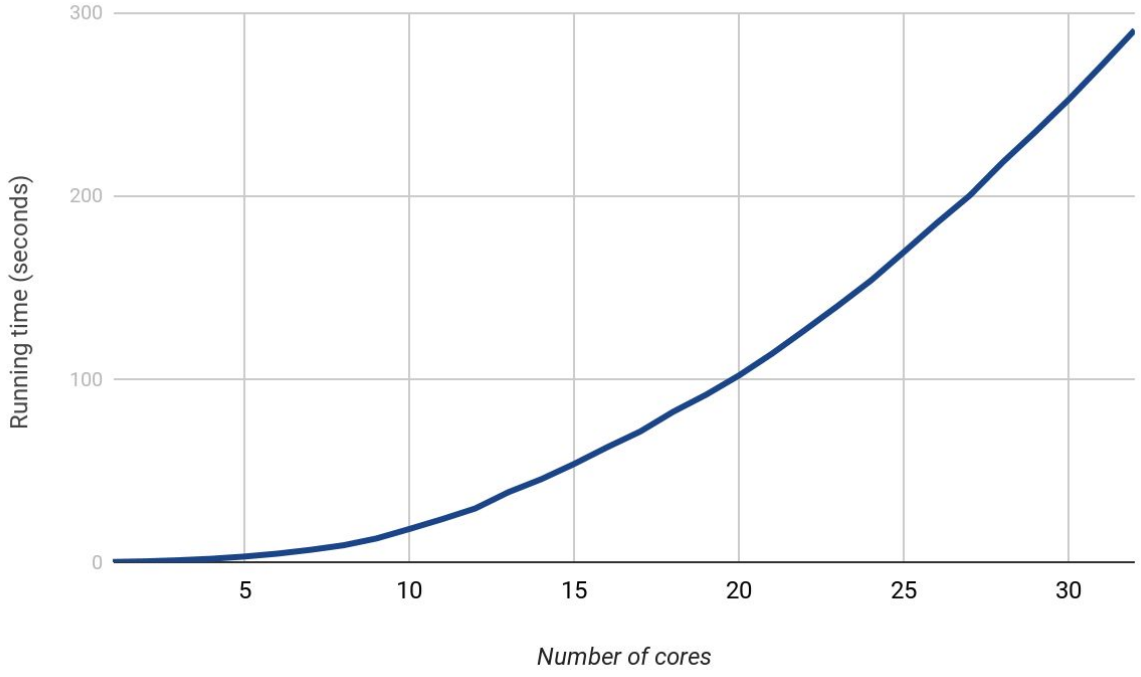


Figure 4: Weak scaling regarding number of processes

b. Strong scaling
Matrix size: 5000 x 5000
Number of nodes: 1
Benchmark data (unit second)

Number of processes	Speedup rate (times)	Run time (seconds)	Serial time (seconds)
1	0.974312902	208.7976599	203.4342539
2	2.114776457	96.19657588	
3	2.617255117	77.72809482	
4	4.367926107	46.57456398	

5	5.687319395	35.76979589	
6	6.202184999	32.80041695	
7	7.263496834	28.00775695	
8	7.414703504	27.43659997	
9	9.739686947	20.88714504	
10	11.14828701	18.24802804	
11	10.96988123	18.54480004	
12	12.05510022	16.87536812	
13	11.03706138	18.43192196	
14	13.13064013	15.49309492	
15	14.74533839	13.79651308	
16	16.37975403	12.41986012	
17	17.98141377	11.31358504	
18	19.80823473	10.27018595	
19	21.96113533	9.263375998	
20	23.42493292	8.684518099	
21	21.49135839	9.465862989	
22	22.17920944	9.172295094	
23	22.57789885	9.010327101	
24	22.28244111	9.129801035	
25	28.65288924	7.099956036	
26	23.89349135	8.514212132	
27	27.01731566	7.529772997	
28	29.88805253	6.806540966	
29	31.83674032	6.38992095	
30	32.02596176	6.352166891	
31	32.78963786	6.204223871	
32	33.58835514	6.056689978	

Visual graph is shown in figure 5

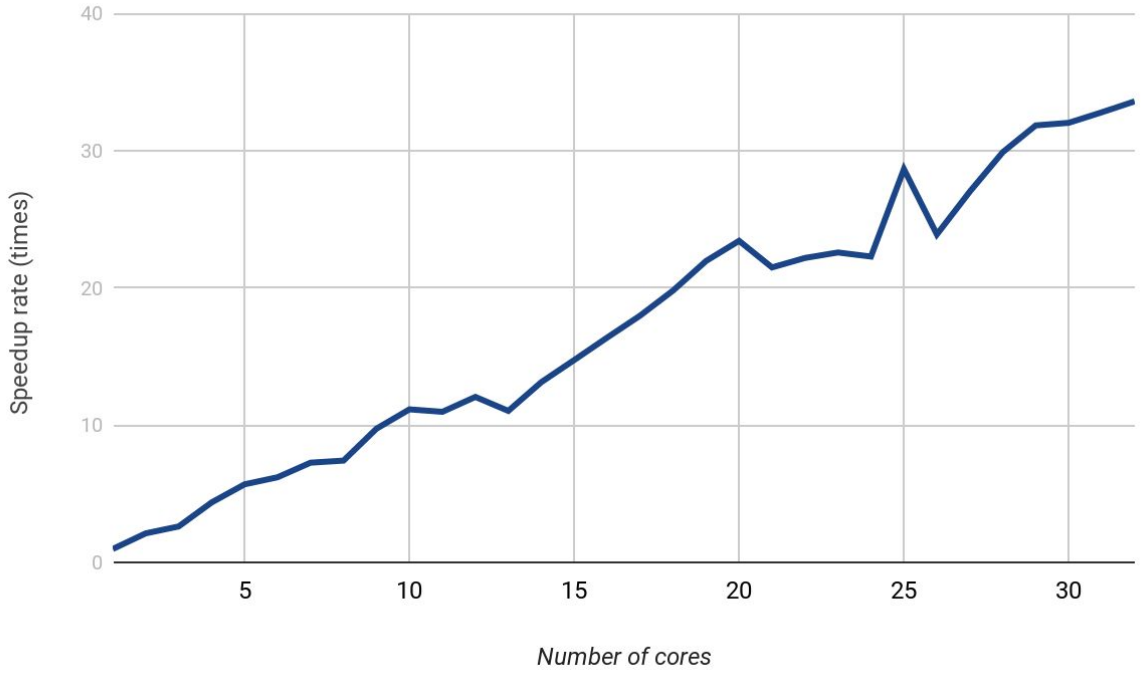


Figure 5: Strong scaling (n = 5000) regarding number of processes

Benchmark regarding number of OMP threads

a. Strong scaling
n = 7000
4 nodes * 4 cores per node
Raw measured data

Number of threads	Speedup rate (times)	Running time (seconds)
1	1	39.11805606
2	1.27600477	30.65666914
3	1.633843084	23.94235802
4	2.084590972	18.7653389
5	2.427073946	16.11737299
6	2.863600627	13.66044402
7	3.283150283	11.91479301
8	3.63686211	10.75599098
9	3.416861015	11.44853592
10	0.6557324005	59.65551806
11	0.4517801245	86.58649182
12	0.288131631	135.7645321

13	0.2740614401	142.734622
14	0.2045571023	191.23294
15	0.1902383967	205.626502
16	0.1164206256	336.0062349

Visual graph is expressed in figure 6.
The performance starts degrading when number of threads reaches 9 threads.
Explanation: Reedbush physical node has 2 sockets, each socket has 18 cores. This means that each node has 36 cores in total. Our configuration is selecting 4 nodes x 4 cores per node. Physical node can not provide enough real parallel thread to each process. Race condition occurs and performance start degrading.
8 threads was little bit better than 9 threads because of memory alignment.

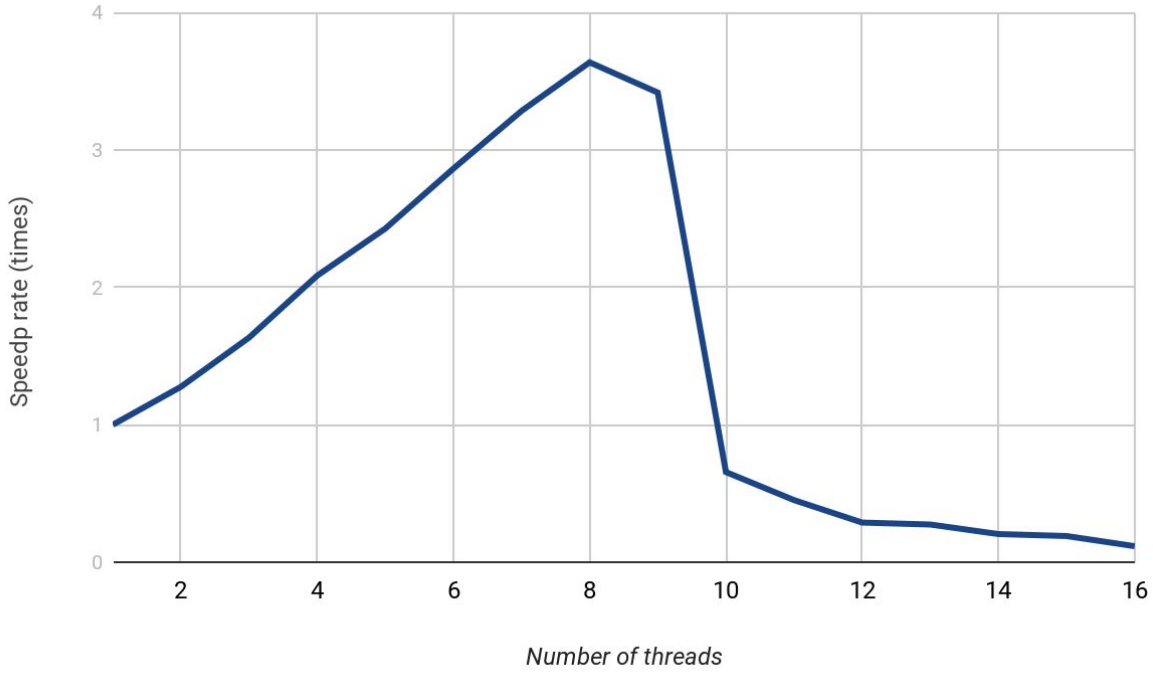


Figure 6. Strong scaling regarding OMP threads

b. Weak scaling
Size 100 for each OMP thread. I.e size of input matrix = number of OMP threads * number of processes * 100
Raw data

Number of threads	Running time (second)
1	0.0928370953
2	0.5520708561
3	1.294378996
4	3.149892092

5	5.030782938
6	8.08631897
7	12.17073202
8	16.06838799
9	25.51082087
10	95.92896199
11	134.1704988
12	153.354568
13	185.7701981
14	258.887589
15	251.8258371
16	280.1839619

Visual graph is shown in figure 7. Similar to the explanation in strong scaling. The scaling rate start degrading rapidly when number of OMP threads surpasses 9 threads.

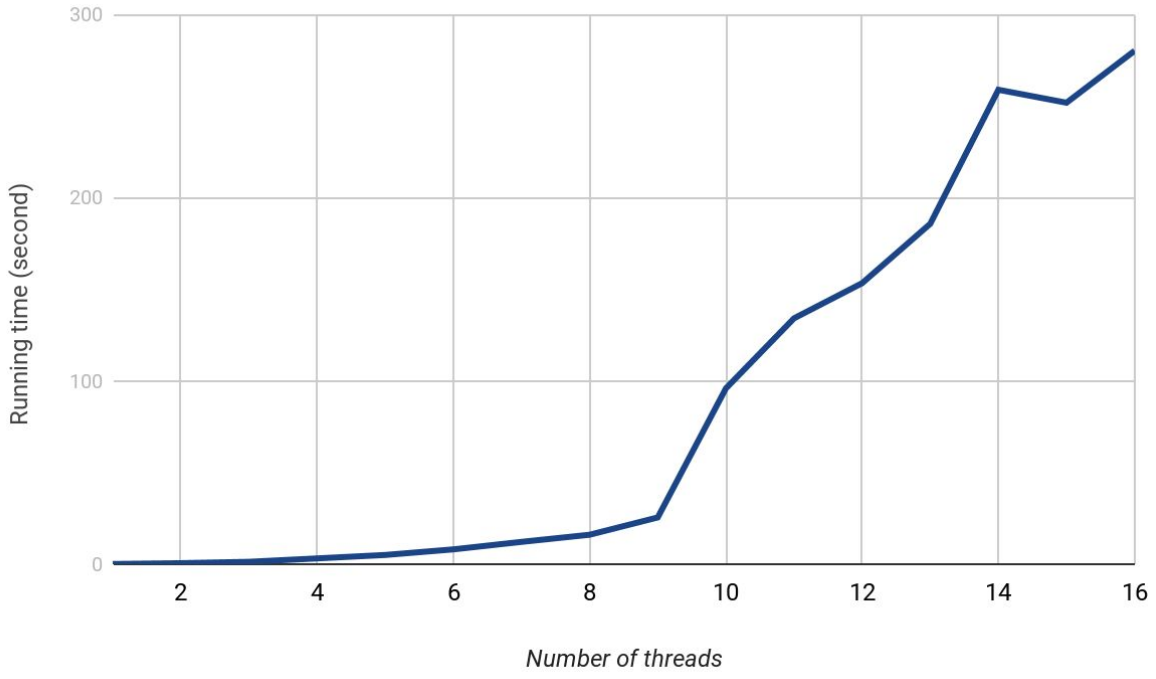


Figure 7. Weak scaling regarding number of OMP threads

c. Try with multiple configuration

Variated configuration to compare performance when distributing calculation over multiple nodes. Visual graph is shown in figure 8.

Number of threads	4 nodes x 4 cores	time	8 nodes x 2 cores	time	2 nodes x 8 cores	time
1	1	39.1312158	1	39.1180560	1	47.5510261

		1		6		1
2	1.277433172	30.63269114	1.27600477	30.65666914	1.359192622	34.98475885
3	1.659329415	23.58254814	1.633843084	23.94235802	1.773542828	26.81132102
4	2.069266933	18.91066599	2.084590972	18.7653389	2.1835205	21.77722907
5	2.47833008	15.78934789	2.427073946	16.11737299	0.4736370831	100.395488
6	2.890672326	13.53706384	2.863600627	13.66044402	0.5570866424	85.35660791
7	3.292953965	11.88331699	3.283150283	11.91479301	0.2789479244	170.4656031
8	3.650715117	10.71878099	3.63686211	10.75599098	0.1025360338	463.749419
9	4.005126632	9.770281792	3.416861015	11.44853592	0.09483585952	501.403439
10	4.4416262	8.810110092	0.6557324005	59.65551806	0.786010014	60.49671793
11	4.749243963	8.239462137	0.4517801245	86.58649182		
12	5.16029731	7.583132029	0.288131631	135.7645321		
13	5.484357799	7.13505888	0.2740614401	142.734622		
14	5.898317019	6.634301901	0.2045571023	191.23294		
15	6.041773726	6.476776123	0.1902383967	205.626502		
16	6.411059555	6.103704929	0.1164206256	336.0062349		

The explanation in strong scaling section is once again confirmed by the fact that in figure 8, with 8 cores per node, speedup rate degrades when number of threads just reaches 5 (5 x 8 = 40 > 38)

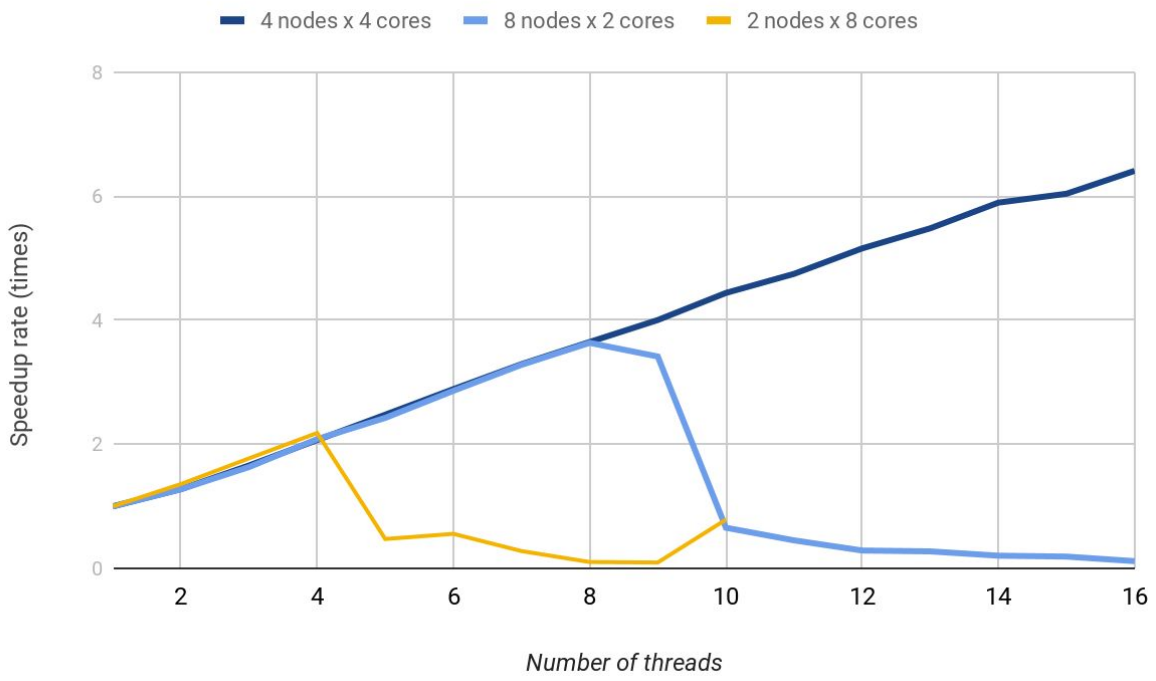


Figure 8: . Variated OpenMP configuration

Amdahl's law analyze

Lets analyze the source code of main algorithm (the code portion is benchmarked and parallelized)
Assume time to init data communication is α
Input matrix: n x n
Number of processes: np
Number of threads: nthreads
Communication speed is: β seconds per 8 byte

Parallelized
O(1) in each processes
	`int i, j, k, tag, receiver, mpi_result, row, skipped_rows_count, sender;`
	`int nrows_local = get_nrows(n, np, rank);`
	`double *local_a = malloc(sizeof(double) * n * nrows_local);`
	`double *local_a_t = malloc(sizeof(double) * n * nrows_local);`
	`double *first_column = malloc(sizeof(double) * n);`
	`double *allgather_buf = malloc(sizeof(double) * n);`
	`assert(local_a != NULL);`
	`assert(local_a_t != NULL);`
	`assert(first_column != NULL);`
	`assert(allgather_buf != NULL);`
	`double tmp, aji;`
	`int nrequests;`
	`if (rank == 0) nrequests = n - nrows_local;`
	`else nrequests = nrows_local;`

	MPI_Request *requests = malloc(sizeof(MPI_Request) * nrequests);
	MPI_Status *statuses = malloc(sizeof(MPI_Status) * nrequests);
	int *displs = malloc(sizeof(int) * np);
	int *recvcounts = malloc(sizeof(int) * np);
	assert(requests != NULL);
	assert(statuses != NULL);
	assert(displs != NULL);
	assert(recvcounts != NULL);

Not parallelized over processes but parallelized over OMP threads
Time complexity $O((\alpha * n + (n / np)^2) / nthreads)$
	//root process
	if (rank == ROOT_RANK){
	j = 0;
	//deliver row data to each other processes from root process
	#pragma omp parallel for private(tag, receiver, k)
	for(i = 0; i < n; i++){
	tag = i / np;
	receiver = i % np;
	if (receiver != 0){
	#pragma omp critical
	k = j++;
	MPI(MPI_Isend(ELM(a, i, 0, n), i + 1, MPI_DOUBLE, receiver, tag, MPI_COMM_WORLD, requests + k));
	}
	}
	//copy to my own
	//(n + np - 1) / np = nrows_local
	#pragma omp parallel for private(row)
	for(i = 0; i < nrows_local; i++){
	row = get_row(np, rank, i);
	memcpy(ELM(local_a, i, 0, n), ELM(a, row, 0, n), sizeof(double) * (row + 1));
	}

Data communication, done parallel
Time complexity
$O(\alpha * (n / np) / nthreads + \beta * n * (n / np))$
	}else {
	//child process
	#pragma omp parallel for private(row, tag)
	for(i = 0; i < nrows_local; i++){
	row = get_row(np, rank, i);
	tag = i;
	MPI(MPI_Irecv(ELM(local_a, i, 0, n), row + 1, MPI_DOUBLE, ROOT_RANK, tag, MPI_COMM_WORLD, requests + i));
	}
	MPI(MPI_Waitall(nrequests, requests, statuses));
	}
	//wait all requests in root process
	if (rank == ROOT_RANK){

	`MPI(MPI_Waitall(nrequests, requests, statuses));` `}`

Parallel
O(n * n / np / nthreads)
	`//transpose to column major`
	`#pragma omp parallel for private(row)`
	`for(i = 0; i < nrows_local; i++){`
	`//also take diagonal elms`
	`row = get_row(np, rank, i);`
	`#pragma omp parallel for`
	`for(j = 0; j <= row; j++){`
	`*ELM(local_a_t, j, i, nrows_local) = *ELM(local_a, i, j, n);`
	`}`
	`}`

	Next will be the most intensive calculation
	START INSIDE `for(i = 0; i < n; i++){`

O(np + β * n + α * np)
Or more exactly
O(np + β * i + α * np)
	`for(j = 0; j < np; j++){`
	`recvcounts[j] = get_nrows(n, np, j) - get_nrows(i, np, j);`
	`displs[j] = j == 0 ? 0 : displs[j - 1] + recvcounts[j -1];`
	`}`
	`//broadcast first column (i.e. first row of column-major) of current iteration`
	`skipped_rows_count = get_nrows(i, np, rank);`
	`MPI(MPI_Allgatherv(ELM(local_a_t, i, skipped_rows_count, nrows_local), recvcounts[rank], MPI_DOUBLE, allgather_buf, recvcounts, displs, MPI_DOUBLE, MPI_COMM_WORLD));`

O(n / nthreads)
More exactly
O(i / nthreads)
	`//put elms of collected buffer into correct order`
	`#pragma omp parallel for private(row)`
	`for(j = 0; j < np; j++){`
	`row = recvcounts[j];`
	`#pragma omp parallel for`
	`for(k = 0; k < row; k++)`
	`first_column[(k + get_nrows(i, np, j)) * np + j] = allgather_buf[displs[j] + k];`
	`}`
	
	`//pre devide first_column to speedup (reduce deviding operation)`
	`row = get_row(np, rank, nrows_local -1);`
	`#pragma omp parallel for`
	`for(j = i + 1; j <= row; j++)`
	`first_column[j] /= first_column[i];`

Most intensive calculation of the whole algorithm. This will be considered in	`//do LDLT calculation`
	`//for all rows`
	`skipped_rows_count = get_nrows(i + 1, np, rank);`
	`//get_nrows of i + 1 because we are going to skip the row i (do j-loop from i+`

the Adamhl's law formula
O(n * n / np / nthreads)
	1 to n)
	`#pragma omp parallel for private(aji, row)`
	`for(j = skipped_rows_count; j < nrows_local; j++){`
	`//backup aji`
	`aji = *ELM(local_a_t, i, j, nrows_local);`
	`row = get_row(np, rank, j);`
	`//first elm`
	`*ELM(local_a_t, i, j, nrows_local) = first_column[row];`
	`//other elms`
	`#pragma omp parallel for`
	`for(k = i + 1; k <= row; k++)`
	`*ELM(local_a_t, k, j, nrows_local) -= aji * first_column[k];`
	`}`
	`MPI(MPI_Barrier(MPI_COMM_WORLD));`
	`}`

	END INSIDE `for(i = 0; i < n; i++){`

Other rest of code does not excess O(n * n / np / nthreads). Hence, we do not analyze more	`//transpose back to row-major`
	`#pragma omp parallel for private(row)`
	`for(i = 0; i < nrows_local; i++){`
	`row = get_row(np, rank, i);`
	`//also take diagonal elms`
	`#pragma omp parallel for`
	`for(j = 0; j <= row; j++){`
	`*ELM(local_a, i, j, n) = *ELM(local_a_t, j, i, nrows_local);`
	`}`
	`if (rank != ROOT_RANK){`
	`//trasfer back to root process`
	`MPI( MPI_Isend(`
	`ELM(local_a, i, 0, n),`
	`row + 1,`
	`MPI_DOUBLE,`
	`ROOT_RANK,`
	`i,`
	`MPI_COMM_WORLD,`
	`requests + i`
	`));`
	`}`
	`}`
	`if (rank == ROOT_RANK){`
	`//receive calculated buffer from all processes`
	`j = 0;`
	`#pragma omp parallel for private(tag, sender, k)`
	`for(i = 0; i < n; i++){`
	`tag = i / np;`

```
      sender = i % np;
      if (sender != 0){
#pragma omp critical
        k = j++;
        MPI(MPI_Irecv(ELM(a, i, 0, n), i + 1, MPI_DOUBLE, sender, tag,
MPI_COMM_WORLD, requests + k));
      }
    }
    //copy from my own
    //(n + np - 1) / np = nrows_local
#pragma omp parallel for private(row)
    for(i = 0; i < nrows_local; i++){
      row = get_row(np, rank, i);
      memcpy(ELM(a, row, 0, n), ELM(local_a, i, 0, n), sizeof(double) * (row +
1));
    }
    MPI(MPI_Waitall(nrequests, requests, statuses));

    /* forward solve L y = b: but y is stored in x
       can be merged to the previous loop */
    for (i=0; i< n; i++) {
      double t = b[i];

#pragma omp parallel for reduction(-:t)
      for (j=0; j< i; j++)
        t -= *ELM(a, i, j, n) * x[j];

      x[i] = t;
    }

    /* backward solve D L^t x = y */
    for (i= n-1; i>= 0; i--) {
      double t = x[i] / *ELM(a, i, i, n);

#pragma omp parallel for reduction(-:t)
      for (j= i+1; j< n; j++)
        t -= *ELM(a, j, i, n) * x[j];

      x[i] = t;
    }
  } else {
    MPI(MPI_Waitall(nrequests, requests, statuses));
  }

  free(statuses);
```

```
  free(local_a);
  free(requests);
  free(first_column);
  free(allgather_buf);
  free(displs);
  free(recvcounts);
  free(local_a_t);
}
```

By the analyze, we observe that the most intensive calculation is parallelized completely due to isolated calculation. Thus, expected speedup ratio is 1/ ((1 - p) + p / s) = np * nthreads