The basic code block that does the computation can be written in two ways as shown in the table below. The code

Table 1: Matrix Multiplication; right is cache-friendly

```
for( i=0; i<NR; i++ ){
  for( j=0; j<NC; j++ ){
    for( k=0; k<NC; k++){
          C[i][j] += A[i][k]*B[k][j];
}
}
}
```

```
for( i=0; i<NR; i++ ){
  for( k=0; k<NC; k++){
    for( j=0; j<NC; j++ ){
          C[i][j] += A[i][k]*B[k][j];
}
}
}
```

on right side should be faster than that in the left column of the table as it is cache-friendly. Table 2 confirms that this is indeed the case. It might not make much difference for small values of `NC` but for larger values the speedup is upto 3. We are going to use cache-friendly computation of matrix multiplication in all files.

Table 2: normal vs. cache-friendly

N	normal	cache-friendly
100	0.008075	0.008205
200	0.063293	0.064621
400	0.594145	0.610361
800	5.386019	4.080133
1000	9.487167	5.573937
2000	87.209641	45.544228
5000	1908.666260	703.574890
8000	8362.995117	2878.790771
10000	17790.554688	6591.408691

Results

Table 3: Time taken. N - matrix size, n - threads/procesess

N	serial	OMP n=2	OMP n=4	OMP n=8	OMP n= 24	MPI n=2	MPI n=4	MPI n=8	MPI n=24
100	0.008	0.009	0.003	0.002	0.007	0.005	0.005	0.003	0.005
200	0.065	0.007	0.042	0.022	0.011	0.059	0.033	0.018	0.023
400	0.610	0.330	0.186	0.106	0.081	0.346	0.251	0.183	0.092
800	4.080	2.195	1.173	0.574	0.486	2.328	1.368	0.696	0.589
1000	5.574	2.842	2.338	0.804	0.594	2.899	2.536	1.166	0.718
2000	45.544	22.061	11.367	5.903	4.065	22.534	11.836	10.047	4.581
5000	703.575	342.611	175.707	90.065	62.475	346.949	179.308	91.602	57.713
8000	2878.791	1400.648	697.816	363.322	234.618	1434.891	715.345	636.908	233.627
10000	6591.409	2737.575	1798.366	740.367	499.030	2812.980	1680.102	877.236	557.603

The following things can be inferred from the results.

1. Optimizing the way core computation of a problem is done can give good speedup. In this case we got a speedup of around 3 by changing the order(row vs. column) in which elements are accessed.
2. Doubling the problem size increases the runtime by 8 times. This is in accordance with the $O(N^3)$ algorithm used.
3. Increasing the number of threads/processes may not always result in proportional increase in speedup. Changing n from 2 to 4 or 4 to 8 increased the speedup by 2 but changing n from 8 to 24 did not increase the speedup by 3. This is mainly due to the increase in overhead associated with thread creation/management.
4. OMP performs slightly better than MPI. (Note: Some of the numbers in Table 3 may not be correct. Some other process running at the same time may have caused this.)
5. The beauty of OMP can be appreciated well in this case. Inserting just a single line in serial code resulted in serious speedup.

Hardware used The machine on which the above results were produced has 24 cores, 15MB L3 cache and it is running Debian OS.

Codes All codes are available at https://github.com/prupro/HPSC/tree/master/2/codes. Files that have 'CF' in the name are to be used.
Serial:

```
#define N 1000
#define NR N
#define NC N

#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

void printMat(float A[NR][NC]);
void initMat(float A[NR][NC],float B[NR][NC]);

int main(){

        static float A[NR][NC];

        static float B[NR][NC];

        static float C[NR][NC] = {{0}}; /* initialize to 0 */

        double start_time, end_time;

        start_time = MPI_Wtime();
        int i,j,k;

        initMat(A,B);           /* fills A with random floats */

        for( i=0; i<NR; i++ ){
                for( k=0; k<NC; k++){
                        for( j=0; j<NC; j++ ){
                                C[i][j] = C[i][j] + A[i][k]*B[k][j];
                        }

                }
        }

        end_time = MPI_Wtime();

        //printMat(C);

        printf("\n Time taken is %f \n",(float)(end_time - start_time));

        return 0;
}


void printMat(float A[NR][NC]){

        int i,j;

        for( i=0; i<NR; i++ ){
```

```
                printf("ROW_%d:",i+1);
                for( j=0; j<NC; j++ ){
                        printf("%.3f\t",A[i][j]);
                }
                printf("\n");
        }

}

void initMat(float A[NR][NC],float B[NR][NC]){

        int i,j;

        for( i=0; i < NR; i++){
                for( j=0; j<NC; j++){
                        A[i][j] = i+j;
                        B[i][j] = i*j;
                }
        }

}
```

OMP:

```
#define N 1000
#define NR N
#define NC N

#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

void printMat(float A[NR][NC]);
void initMat(float A[NR][NC],float B[NR][NC]);

int main(){

        static float A[NR][NC];

        static float B[NR][NC];

        static float C[NR][NC] = {{0}}; /* initialize to 0 */

        double start_time, end_time;

        start_time = MPI_Wtime();
        int i,j,k;

        initMat(A,B);           /* fills A with random floats */
#pragma omp parallel for private(j,k)
        for( i=0; i<NR; i++ ){
                for( k=0; k<NC; k++){
                        for( j=0; j<NC; j++ ){
                                C[i][j] = C[i][j] + A[i][k]*B[k][j];
                        }

                }
        }

        end_time = MPI_Wtime();

        //printMat(C);

        printf("\n Time taken is %f \n",(float)(end_time - start_time));

        return 0;
}


void printMat(float A[NR][NC]){

        int i,j;

        for( i=0; i<NR; i++ ){
                printf("ROW %d:",i+1);
                for( j=0; j<NC; j++ ){
```

```
                        printf("%.3f\t",A[i][j]);
                }
                printf("\n");
        }

}

void initMat(float A[NR][NC],float B[NR][NC]){

        int i,j;

        for( i=0; i < NR; i++){
                for( j=0; j<NC; j++){
                        A[i][j] = i+j;
                        B[i][j] = i*j;
                }
        }

}
```

MPI:

```
#define N 1000
#define NR N
#define NC N
#define MASTER 0
#define TOMASTER 8055
#define TOSLAVE 5088 /* slave-slave connection is not required */


#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

void printMat(float A[NR][NC]);
void initMat(float A[NR][NC],float B[NR][NC]);

int main(int argc, char **argv){

        double start_time, end_time;
        int i,j,k;
        int npes;
        int mype;
        int rows;

        MPI_Status status;

        MPI_Init(&argc, &argv);

        MPI_Comm_size(MPI_COMM_WORLD, &npes);
        MPI_Comm_rank(MPI_COMM_WORLD, &mype);

        if ( npes < 2 ) {
                printf("Should␣have␣at␣least␣2␣PEs\n");
                MPI_Finalize();
                exit(1);
        }

        start_time = MPI_Wtime();

        static float A[NR][NC];
        static float B[NR][NC]; /*All PEs need B*/
        static float C[NR][NC] = {{0}}; /* initialize to 0 */

        if ( mype == MASTER ){


                initMat(A,B);           /* fills A with random floats */

                int noOfRows[npes]; /* no of rows computed by a PE */
                int startingRow[npes]; /* */
                startingRow[0] = 0;
                noOfRows[0] = NR/(1*npes);
                int rowsRemaining = NR - noOfRows[0];
```

```
                int l;
                int rem = rowsRemaining%(npes-1);

                for(l=1;l<npes;l++){
                        noOfRows[l] = rowsRemaining /(npes-1);
                        if( rem > 0){
                                noOfRows[l]++;
                                rem--;
                        }
                        startingRow[l] = startingRow[l-1] + noOfRows[l-1];

                }

                for( l = 1; l<npes; l++){  /* send rows to PEs*/
                        rows = noOfRows[l];

                        MPI_Send(&rows,1,MPI_INT,l,TOSLAVE,MPI_COMM_WORLD);
                        MPI_Send(&A[startingRow[l]][0],rows*NC,MPI_FLOAT,l,
                                        TOSLAVE,MPI_COMM_WORLD);
                        MPI_Send(B,NR*NC,MPI_FLOAT,l,
                                        TOSLAVE,MPI_COMM_WORLD);

                }


                for( i=0; i<noOfRows[0]; i++ ){ /* computation */
                        for( k=0; k<NC; k++){
                                for( j=0; j<NC; j++ ){
                                        C[i][j] = C[i][j] + A[i][k]*B[k][j];
                                }

                        }
                }

                for( l = 1; l<npes; l++){  /* receive rows from PEs*/
                        rows = noOfRows[l];

                        MPI_Recv(&C[startingRow[l]][0],rows*NC,MPI_FLOAT,l,
                                        TOMASTER,MPI_COMM_WORLD,MPI_STATUS_IGNORE);

                }

                end_time = MPI_Wtime();

                //printMat(C);

                printf("\n Time taken is %f \n",(float)(end_time - start_time));
        }

        else {
                MPI_Recv(&rows,1,MPI_INT,MASTER,TOSLAVE,MPI_COMM_WORLD,MPI_STATUS_IGNORE);

                MPI_Recv(A,rows*NC,MPI_FLOAT,MASTER,TOSLAVE,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
```

```
                MPI_Recv(B,NR*NC,MPI_FLOAT,MASTER,TOSLAVE,MPI_COMM_WORLD,MPI_STATUS_IGNORE);

                for( i=0; i<rows; i++ ){ /* computation */
                        for( k=0; k<NC; k++){
                                for( j=0; j<NC; j++ ){
                                        C[i][j] = C[i][j] + A[i][k]*B[k][j];
                                }

                        }
                }

                MPI_Send(C,rows*NC,MPI_FLOAT,MASTER,
                                TOMASTER,MPI_COMM_WORLD);

        }

        MPI_Finalize();
        return 0;
}


void printMat(float A[NR][NC]){

        int i,j;

        for( i=0; i<NR; i++ ){
                printf("ROW %d:",i+1);
                for( j=0; j<NC; j++ ){
                        printf("%.3f\t",A[i][j]);
                }
                printf("\n");
        }

}

void initMat(float A[NR][NC],float B[NR][NC]){

        int i,j;

        for( i=0; i < NR; i++){
                for( j=0; j<NC; j++){
                        A[i][j] = i+j;
                        B[i][j] = i*j;
                }
        }

}
```