Improving scalability of matrix multiplication

CPU SPECS:

```
GPU: 1xTesla K80 , having 2496 CUDA cores, compute 3.7,  12GB(11.439GB Usable) GDDR5  VRAM

CPU: 1xsingle core hyper threaded i.e(1 core, 2 threads) Xeon Processors @2.3Ghz (No Turbo Boost) , 45MB Cache

RAM: ~12.6 GB Available

Disk: ~320 GB Available
```

CODE:

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

void print_matrix(double*,int);
void fill_matrix(double*,int,int);
void fill_identity_matrix(double*,int);
void multiply(double*,double*,double*,int);
void generate(double**, double**, double**, int);
void distribute(double*, double*, int, int,MPI_Datatype*,  MPI_Comm, MPI_Request*);

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

 int wrank, nproc;
 double start, end;

 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &nproc);
 MPI_Comm_rank(MPI_COMM_WORLD, &wrank);

 if(argc < 2 && wrank == 0) {
  printf("Usage: mpirun -np 4 ./fox n (where n is the matrix size)\n");
  MPI_Finalize();
  return -1;
 }

 if(argc < 2 && wrank != 0) {
  MPI_Finalize();
  return -1;
 }

 double a; int b;
 a = sqrt(atoi(argv[1]));
 b = a;

 int n = atoi(argv[1]);

 MPI_Status stat;
 int myid, rowrank, colrank;
 MPI_Comm proc_grid, proc_row, proc_col;
```

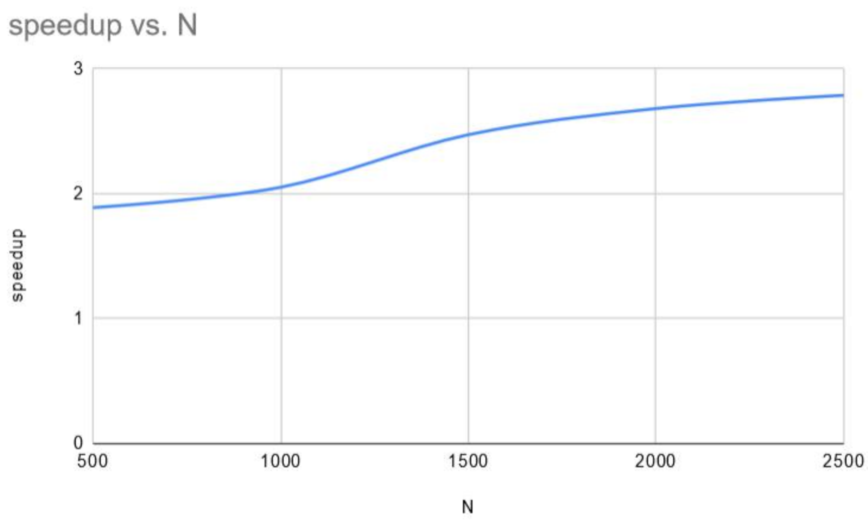
```
MPI_Datatype newtype;
int i,j;
MPI_Request request;
MPI_Status status;

int ndim=2,dims[2]={0,0};

int coords[2];

int reorder=1;
int periods[2]={0,0};

MPI_Dims_create(nproc,ndim,dims);
MPI_Cart_create(MPI_COMM_WORLD,ndim,dims,periods,reorder,&proc_grid);
MPI_Comm_rank(proc_grid,&myid);
int pn = dims[0];

int blocksize = n*n/(pn*pn);
double *Ab=(double *)calloc(blocksize,sizeof(double));
double *Bb=(double *)calloc(blocksize,sizeof(double));
double *Ab_temp=(double *)calloc(blocksize,sizeof(double));
double *Bb_temp=(double *)calloc(blocksize,sizeof(double));
double *Cb=(double *)calloc(blocksize,sizeof(double));
double *C, *A, *B;

if(myid == 0) {
  generate(&A,&B,&C, n);
  start = MPI_Wtime();
  distribute(A,B, n, pn, &newtype,  proc_grid, &request);
}

MPI_Cart_coords(proc_grid,myid,ndim,coords);
MPI_Recv(Ab, blocksize, MPI_DOUBLE, 0, 111, proc_grid, &status);
MPI_Recv(Bb, blocksize, MPI_DOUBLE, 0, 222, proc_grid, &status);


MPI_Comm_split(proc_grid,coords[0],coords[1],&proc_row);
MPI_Comm_rank(proc_row,&rowrank);

MPI_Comm_split(proc_grid,coords[1],coords[1],&proc_col);
MPI_Comm_rank(proc_col,&colrank);
MPI_Request req1, req2;
int k, m;
for(k = 0; k < pn; ++k) {
  m = (coords[0]+k) % pn;
  memcpy(Ab_temp, Ab, sizeof(double)*blocksize);
  MPI_Bcast(Ab_temp, blocksize, MPI_DOUBLE, m, proc_row);
  MPI_Irecv(Bb_temp, blocksize, MPI_DOUBLE, colrank == pn-1 ? 0 : colrank+1, 333, proc_col, &req1);
  MPI_Isend(Bb, blocksize, MPI_DOUBLE, colrank == 0 ? pn-1 : colrank-1, 333, proc_col, &req2);
  multiply(Ab_temp, Bb, Cb, n/pn);
  MPI_Wait(&req1, &status);
  MPI_Wait(&req2, &status);
  memcpy(Bb, Bb_temp, sizeof(double)*blocksize);

}

MPI_Isend(Cb, blocksize, MPI_DOUBLE, 0, 444, proc_grid, &request);
```

```
  if(myid == 0) {
    for(i = 0; i < pn*pn; ++i) {
      MPI_Probe(MPI_ANY_SOURCE, 444, proc_grid, &status);
      MPI_Cart_coords(proc_grid,status.MPI_SOURCE,ndim,coords);
      MPI_Recv(&C[(n*(n*coords[0]+coords[1]))/pn], 1, newtype, status.MPI_SOURCE, 444, proc_grid,
&status);
      end = MPI_Wtime();

    }
    printf("Result Matrix C is:\n");
    print_matrix(C, n);
    free(A);
    free(B);
    free(C);
  }
  MPI_Wait(&request,&status);
  free(Ab);
  free(Ab_temp);
  free(Bb);
  free(Bb_temp);
  free(Cb);
  MPI_Finalize();

  return 0;
}

void fill_matrix(double *m, int n, int seed) {
  int row, col;
  srand(seed);
  for (row=0; row<n;row++)
    for (col=0; col<n;col++)
      m[row*n+col] = ((double)rand()*1000/(double)(RAND_MAX));
}

void fill_identity_matrix(double *m, int n) {
  int row, col;
  for (row=0; row<n;row++)
    for (col=0; col<n;col++)
      m[row*n+col] = row == col ? 1 : 0;
}

void print_matrix(double *m, int n) {
  int row, col;
  for (row=0; row<n;row++){
    for (col=0; col<n;col++){
      printf("%f ", m[row*n+col]);
    }
    printf("\n");
  }
  printf("\n");
}

void multiply(double *a,double *b, double *c, int n) {
  int k,j,i;
  for(k = 0; k < n; ++k)
    for(i = 0; i < n; ++i)
      for(j = 0; j < n; ++j)
```

```
        c[i*n+j] += a[i*n+k]*b[k*n+j];
}

void generate(double** A, double** B, double** C, int n){
 *A=(double *)calloc(n*n,sizeof(double));
 fill_matrix(*A, n, 23);
 printf("\nMatrix A is:\n");
 print_matrix(*A, n);

 *B=(double *)calloc(n*n,sizeof(double));
 fill_identity_matrix(*B,n);
 printf("Matrix B is:\n");
 print_matrix(*B, n);

 *C=(double *)calloc(n*n,sizeof(double));
}

void distribute(double* A, double* B, int n, int pn, MPI_Datatype *newtype,  MPI_Comm proc_grid,
MPI_Request *request){
 int count=n/pn; int blocklen=n/pn; int stride=n; int grank;
 MPI_Type_vector(count,blocklen,stride,MPI_DOUBLE,newtype);
 MPI_Type_commit(newtype);
 int pos[2], i, j;

 for(i = 0; i < pn; ++i) {
   for(j = 0; j < pn; ++j) {
     pos[0] = i; pos[1] = j;
     MPI_Cart_rank(proc_grid,pos,&grank);
     MPI_Isend(&A[(n*(n*i+j))/pn], 1, *newtype, grank, 111, proc_grid, request);
     MPI_Isend(&B[(n*(n*i+j))/pn], 1, *newtype, grank, 222, proc_grid, request);
   }
 }
}
```

Analysis of above code for keeping number of process contant i.e 2 and increasing number size of matrix:

N	speedup
500	1.888889
1000	2.051587
1500	2.472356
2000	2.6817
2500	2.787816



Methodology:

Previously in matrix multiplication we were sending each row of matrix A to respective number of processes instead of that in above code mentioned we are using block matrix multiplication or Fox's algorithm for matrix multiplication. In this algorithm we divide matrix in submatrix or blocks.

Conclusion:

From the above analysis we can conclude that by increasing the size of matrix we can achieve more speedup even though we only 2 parallel processes