SUMMER SCHOOL PROJECT
Implementation of MPI-based conjugate gradient method for solving the Poisson equation on Cartesian grid

Van-Dang NGUYEN* and Alok JUNEJA†
Tutor: Michael HANKE ‡

October 7, 2015

Abstract

In this project, we develop an MPI-based conjugate gradient method (CG) [4] to solve the Poisson equation on Cartesian grids. The finite difference method will be used for space discretization. Thanks to the funtionalities of the MPI virtual topology [1], the computational domain is decomposed into subdomains and then each subdomain is assigned to an MPI process. The performance analysis will also be taken into account in this project.

1 Poisson equation

We consider a Poisson equation which has the following form

$$\nabla^2 u(\boldsymbol{x}) = f(\boldsymbol{x}) \text{ in } \Omega \tag{1}$$

and is subjected to the boundary conditions

$$u(\boldsymbol{x}) = u_0(\boldsymbol{x}) \text{ on } \partial\Omega. \tag{2}$$

where $\boldsymbol{x} = (x_i)_{i=1\,\ldots\,d}$ is the spatial variable and d is the problem dimension.
In this project, we only focus on the computational domain which is a box $\Omega = \Pi_{i=1}^{d}[a_i, b_i]$.

2 Conjugate gradient method for solving the Poisson equation

For simplification, we discribe how to solve the 1D equation using finite difference method. We first discretize the computational domain $\Omega = [a_1, b_1]$ into $N_x + 1$ equal subintervals $[x_i, x_{i+1}]$ by using $N_x + 2$ points $x_0 = a_1, x_1 = a_1 + h_x, \ldots, x_{N_x+1} = b_1$, where $h_x = \frac{b_1 - a_1}{N_x+1}$.

*HPCViz, CSC, KTH, dang.1032170@gmail.com
†Stockholm University, junejaalok@gmail.com
‡KTH, hanke@kth.se

We then replace the second-order derivatives with second-order finite difference approximations

$$\nabla^2 u(\boldsymbol{x}) = u''(x) = \frac{u_{i+1} - 2\,u_i + u_{i-1}}{h_x^2} + \mathcal{O}(h_x^2) \tag{3}$$

Eq. 1 for 1D now becomes

$$\frac{u_{i+1} - 2\,u_i + u_{i-1}}{h_x^2} = f_i \tag{4}$$

which is equivalent to the matrix form

$$\mathbf{A}\mathbf{u} = \mathbf{f} \tag{5}$$

After applying the boundary conditions, we can obtain a numerical solution u by multiplying both sides of equation with the inverse of matrix $\mathbf{A}$,

$$\mathbf{u} = \mathbf{A}^{-1}\mathbf{f} \tag{6}$$

Direct computation of $\mathbf{A}^{-1}$ is time-consuming and memory-demending, especially for higher dimensions. So, researchers try to develop iterative methods which can overcome these two drawbacks of the direct method.

In the scope of this project, we will only focus on the conjugate gradient method (CG) [4] which was developed for particular systems of linear equations whose matrix is symmetric and positive-definite. It is especially suitable for the Poisson equation that we mentioned above. The simplest version which is iterative and has no preconditioning is considered. The method is described in algorithm 1.

Algorithm 1 The conjugate gradient method for solving the symmetric positive definite system $\mathbf{A}\mathbf{u} = \mathbf{f}$, starting with $\mathbf{u}^0$

procedure CG
  $\mathbf{r}^0 := \mathbf{f} - \mathbf{A}\mathbf{u}^0$
  $\mathbf{p}^0 := \mathbf{r}^0$
  $k := 0$
Repeat
  $\alpha_k := \dfrac{(\mathbf{r}^k)^T\mathbf{r}^k}{(\mathbf{p}^k)^T\mathbf{A}\mathbf{p}^k}$
  $\mathbf{u}^{k+1} := \mathbf{u}^k + \alpha_k\mathbf{p}^k$
  $\mathbf{r}^{k+1} := \mathbf{r}^k - \alpha_k\mathbf{A}\mathbf{p}^k$
  if $\mathbf{r}^{k+1}$ is small enough or k is too large **then** exit
  $\beta_k := \dfrac{(\mathbf{r}^{k+1})^T\mathbf{r}_{k+1}}{(\mathbf{r}^k)^T\mathbf{r}^k}$
  $\mathbf{p}^{k+1} := \mathbf{r}^{k+1} + \beta_k\mathbf{p}^k$.
  $k = k + 1$
End repeat
Return $\mathbf{u}^{k+1}$

When using the finite difference discretization for the Poisson equation, however, the computation of $\mathbf{A}\mathbf{u}$ is simpified. $\mathbf{A}\mathbf{u}$ for 2D

```
Au[INDEX_2D_to_1D(i,j)] =coeff[1]*(u[INDEX_2D_to_1D(i+1,j)]-2.0*u[INDEX_2D_to_1D(i,j
    ↪ )]+u[INDEX_2D_to_1D(i-1,j)])+coeff[2]*(u[INDEX_2D_to_1D(i,j+1)]-2.0*u[
    ↪ INDEX_2D_to_1D(i,j)]+u[INDEX_2D_to_1D(i,j-1)]);
```

and for 3D, **Au** is

```
Au[INDEX_3D_to_1D(i,j,k)]=coeff[1]*(u[INDEX_3D_to_1D(i+1,j,k)]-2.0*u[INDEX_3D_to_1D(
    ↪ i,j,k)]+u[INDEX_3D_to_1D(i-1,j,k)])+coeff[2]*(u[INDEX_3D_to_1D(i,j+1,k)]-2.0*
    ↪ u[INDEX_3D_to_1D(i,j,k)]+u[INDEX_3D_to_1D(i,j-1,k)])+coeff[3]*(u[
    ↪ INDEX_3D_to_1D(i,j,k+1)]-2.0*u[INDEX_3D_to_1D(i,j,k)]+u[INDEX_3D_to_1D(i,j,k
    ↪ -1)]);
```

where `INDEX_2D_to_1D` and `INDEX_3D_to_1D` are formulas used to define 1D indices when tranforming 2D and 3D to 1D arrrays respectively. We will discuss these two formulas later.

An MPI version of this method is exactly the same on each MPI process except that before starting an iteration, $\mathbf{p}^k$ needs to be updated by MPI communication `MPI_Sendrecv`. The collective communication `MPI_Allreduce` is used some times to get global information, i.e for the use of all MPI processes (see algorithm 2).

Algorithm 2 An MPI version of the CG method

1: **procedure** CG_MPI
2: Input $\mathbf{u}^0$, `MAX_ITERATOR` and ERROR TOLERANCE `eps`
3: $\mathbf{r}^0 := \mathbf{f} - \mathbf{A}\mathbf{u}^0$
4: $\mathbf{p}^0 := \mathbf{r}^0$
5: $k := 0$
6: `rTr_local`:= $(\mathbf{r}^0)^T\mathbf{r}^0$
7: `rTr_prev`:= sum(`rTr_local`), by using `MPI_Allreduce`.
8: *Repeat*
9: Update $\mathbf{p}^k$ by using `MPI_Sendrecv`
10: `pTAp_local`:= $(\mathbf{p}^k)^T\mathbf{A}\mathbf{p}^k$
11: `pTAp`:= sum(`pTAp_local`), by using `MPI_Allreduce`.
12: α_k :=`rTr_prev/pTAp`
13: $\mathbf{u}^{k+1} := \mathbf{u}^k + \alpha_k\mathbf{p}^k$
14: $\mathbf{r}^{k+1} := \mathbf{r}^k - \alpha_k\mathbf{A}\mathbf{p}^k$
15: `rTr_local`:= $(\mathbf{r}^{k+1})^T\mathbf{r}^{k+1}$
16: `rTr`:= sum(`rTr_local`), by using `MPI_Allreduce`.
17: **if** (`rTr<eps`) or (`k>MAX_ITERATOR`) **then** exit
18: β_k :=`rTr/rTr_prev`
19: `rTr_prev`:=`rTr`
20: $\mathbf{p}^{k+1} := \mathbf{r}^{k+1} + \beta_k\mathbf{p}^k$.
21: $k = k + 1$
22: *End repeat*
23: Return $\mathbf{u}^{k+1}$

3 Implementation

3.1 Space discretization using MPI virtual topology

MPI provides an efficient way to create Cartersian grids by using three functions `MPI_Dims_create`, `MPI_Cart_create` and `MPI_Cart_coords` [1].

`MPI_Dims_create` automatically selects a balanced distribution of processes per coordinate direction.

`MPI_Cart_create` makes a new communicator to which topology information has been attached

`MPI_Cart_coords` determines process coords in cartesian topology given rank in group

```
int dims[ndims];
int periods[ndims];
const int reorder=false;
for (i=0;i<ndims;i++)
{
  dims[i]    = 0;
  periods[i] = false;
}
MPI_Dims_create(nprocs, ndims, dims);
MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, periods, reorder, &mycomm);
int coords[ndims];
MPI_Cart_coords(mycomm, rank, ndims, coords);
```

On each MPI process, the global indices are defined

```
xid_start = (coords[0]*NX)/dims[0]+1;
xid_end = ((coords[0]+1)*NX)/dims[0];
yid_start = (coords[1]*NY)/dims[1]+1;
yid_end = ((coords[1]+1)*NY)/dims[1];
zid_start = (coords[2]*NZ)/dims[2]+1;
zid_end = ((coords[2]+1)*NZ)/dims[2];
```

Fig. 1 shows one example where the computational domain is decomposed to 4 subdomains corresponding to 4 MPI processes P0, P1, P2 and P3. Here we choose $NX = NY = 5$, the contribution of the subdomains on MPI processes looks like

```
Process 0 manages indices from 0 to 3 in x-direction and from 0 to 3 in y-direction.
Process 1 manages indices from 0 to 3 in x-direction and from 2 to 6 in y-direction.
Process 2 manages indices from 2 to 6 in x-direction and from 0 to 3 in y-direction.
Process 3 manages indices from 2 to 6 in x-direction and from 2 to 6 in y-direction.
```

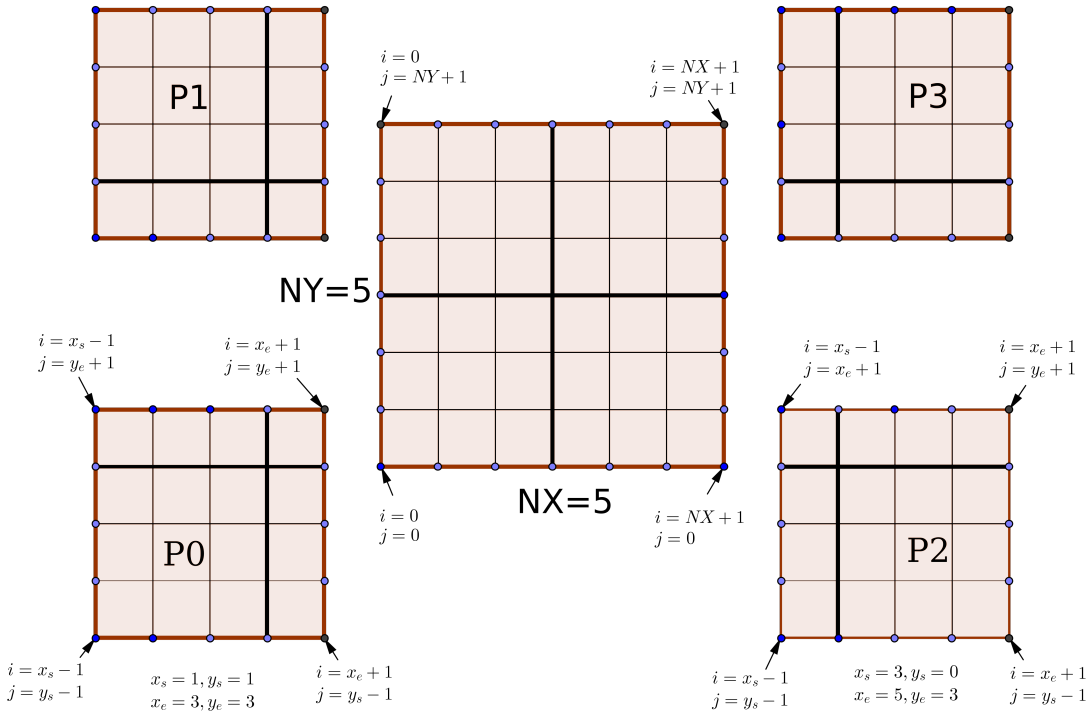


Figure 1: A computational domain (center) is decomposed into four subdomains corresponding to four MPI processes P0, P1, P2 and P3.

We note that, on each subdomain ghost rows and columns are added for data exchange. In subdomain managed by P0 for instance, we add one more row $j = y_e + 1$ and one more column $i = x_e + 1$. During the communication, ghost column corresponding to $i = x_e + 1$ (in P0) will receive data from column corresponding to $i = x_s$ (in P2), ghost row corresponding to $j = y_e + 1$ (in P0) will receive data from column corresponding to $j = y_s$ (in P1).

To do communication between processes, we need to find the neighbours of each MPI process. In 2D, except the subdomains near the boundary, each subdomain has four neighbours: left, right, front, behind (see Fig. 2).

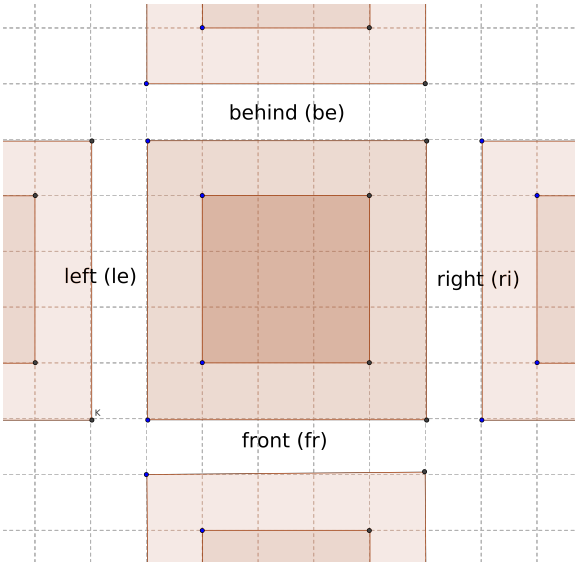


Figure 2: A subdomain (center) has four neighbours: left (le), right (ri), front (fr), and behind (be).

This work can be done automatically by using `MPI_Cart_shift`

```
MPI_Cart_shift(mycomm, 0, 1, &(neighbour[le]), &(neighbour[ri]));
MPI_Cart_shift(mycomm, 1, 1, &(neighbour[fr]), &(neighbour[be]));
```

These commands applied for domain in Fig. 1 will result

```
Process 0 has four neighbours : left -2 behind 1 right 2 front -2
Process 1 has four neighbours : left -2 behind -2 right 3 front 0
Process 2 has four neighbours : left 0 behind 3 right -2 front -2
Process 3 has four neighbours : left 1 behind -2 right -2 front 2
```

We note that the negative process means there is no process in respective direction.

3.2 Data re-organizing

To simplify the data communication between processes, we reorganize data from 3D and 2D arrays to 1D array. Assume that i, j, k is the index of the matrix in x-, y-, and z- directions respectively, we choose to compute the indices for 1D by the following formulas

```
INDEX_2D_to_1D(i,j) ((j-(yid_start-1))*(xid_end-xid_start+3)+i-(xid_start-1))
```

```
INDEX_3D_to_1D(i,j,k) ((k-(zid_start-1))*(yid_end-yid_start+3)*(xid_end-xid_start+3)
    ↪ +(j-(yid_start-1))*(xid_end-xid_start+3)+i-(xid_start-1))
```

3.3 Memory allocation

Since the data structure is reorganized from 2D and 3D to 1D, we only need to allocate 1D array which can be done by using `calloc`. Although the global indices are defined, the 1D array needs to be locally allocated to save memory. In 2D

```
u = calloc( (xid_end-xid_start+3) * (yid_end-yid_start+3), sizeof(double));
```

In 3D

```
u = calloc( (xid_end-xid_start+3) * (yid_end-yid_start+3) * (zid_end-zid_start+3),
    ↪ sizeof(double));
```

3.4 Communication

3.4.1 Derived data types

In 2D, we have to exchange data between processes along the interfaces which belongs to rows or columns. The data is contiguous along $x-$direction with $x_e - x_s + 3$ elements. However, when the Dirichlet boundary conditions [2] are applied, the number of elements is reduced to $x_e - x_s + 1$. So, the data type can be built as the follows

```
MPI_Type_contiguous( xid_end-xid_start+1, MPI_DOUBLE, &xslice_type);
MPI_Type_commit(&xslice_type);
```

After applying the boundary conditions, the data along $y-$direction has $y_e - y_s + 1$ elements but they are totally seperated by a stride of $x_e - x_s + 3$

```
MPI_Type_vector(yid_end-yid_start+1,1,xid_end-xid_start+3,MPI_DOUBLE,&yslice_type);
MPI_Type_commit(&yslice_type);
```

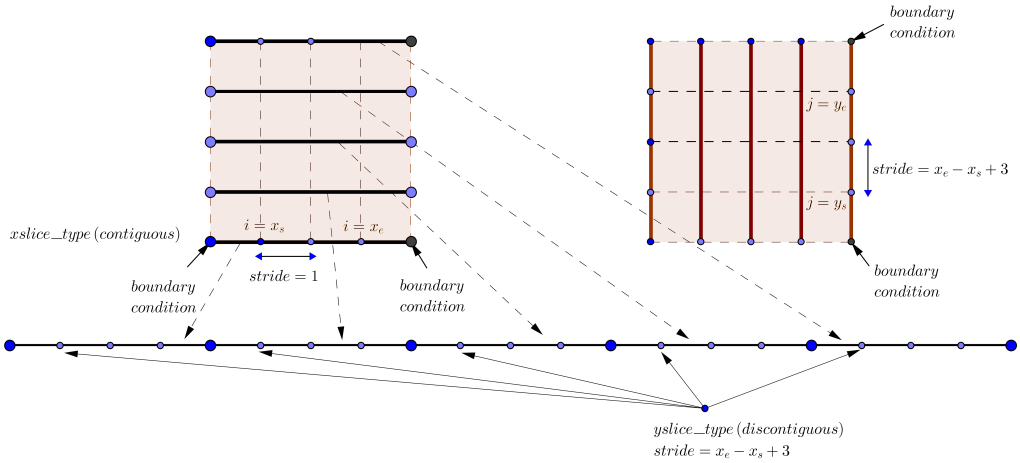


Figure 3: Derived data type for 2D problem. After applying the boundary conditions, the data is contiguous along $x-$direction with $x_e - x_s + 1$ elements. Along $y-$direction there are $y_e - y_s + 1$ elements but they are totally seperated by a stride of $x_e - x_s + 3$.

In 3D, we have to exchange 2D data slab between processes along the interface. The data on $xy-$plane is contiguous. However, the boundary conditions exclude two columns and two rows from the data plane. So, it becomes piecewise contiguous and the stride between any two closest contiguous pieces of data is $x_e - x_s + 3$. There are $y_e - y_s + 1$ pieces of contiguous data. The data needs to exchange between $xy-$planes can be derived as the follows (Fig. 4)

```
MPI_Type_vector( yid_end-yid_start+1, xid_end-xid_start+1, xid_end-xid_start+3 ,
    ↪ MPI_DOUBLE, &xyslice_type);
MPI_Type_commit(&xyslice_type);
```

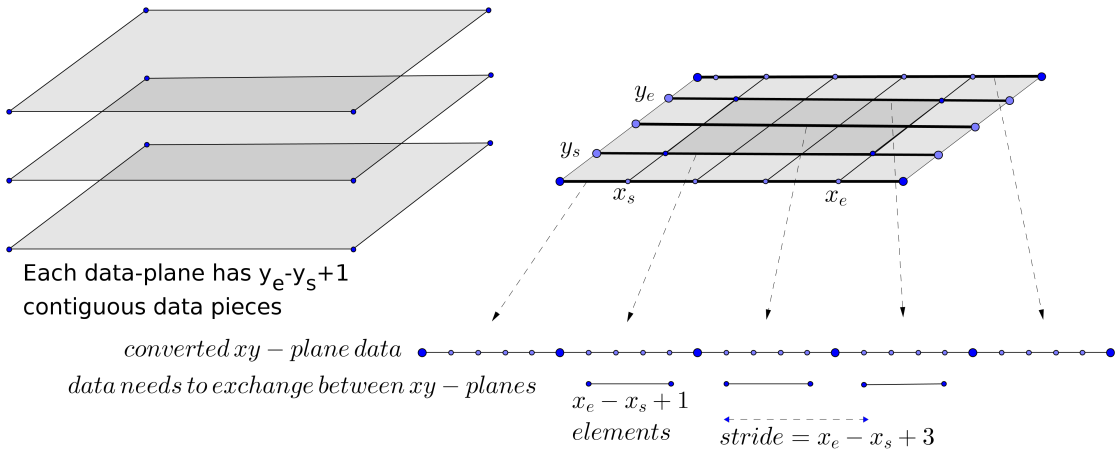


Figure 4: The data exchange between $xy-$planes.

The data on $zx-$plane is piecewise contiguous. Each contiguous piece of data has $x_e - x_s + 1$ elements. The distance between any two closest contiguous pieces of data is $(x_e - x_s + 3)(y_e - y_s + 3)$.

```
MPI_Type_vector(zid_end-zid_start+1,xid_end-xid_start+1, (xid_end-xid_start+3)*(
    ↪ yid_end-yid_start+3), MPI_DOUBLE, &zxslice_type);
MPI_Type_commit(&zxslice_type);
```

The data on $yz-$plane is totally seperated and any two closest elements have the same distance of $x_e - x_s + 3$.

```
MPI_Type_vector( (yid_end-yid_start+3)*(zid_end-zid_start+3), 1, xid_end-xid_start
    ↪ +3, MPI_DOUBLE, &yzslice_type);
MPI_Type_commit(&yzslice_type);
```

3.4.2 Data exchange

In 2D, an MPI process sends and receives data in column, i.e `yslice_type` to/from their left and right neighbours whereas the data in row, i.e `xslice_type` when exchanging with front and behind

neighbours. The communication is done by the following code (see Fig. 5)

```
MPI_Sendrecv(&(u[INDEX_2D_to_1D(xid_start, yid_start)]), 1, yslice_type, neighbour[
    ↪ le], tag, &(u[INDEX_2D_to_1D(xid_end+1, yid_start)]), 1, yslice_type,
    ↪ neighbour[ri], tag, mycomm, &statut);
MPI_Sendrecv(&(u[INDEX_2D_to_1D(xid_end, yid_start)]), 1, yslice_type, neighbour[ri
    ↪ ], tag,&(u[INDEX_2D_to_1D(xid_start-1, yid_start)]), 1, yslice_type,
    ↪ neighbour[le], tag, mycomm, &statut);
MPI_Sendrecv(&(u[INDEX_2D_to_1D(xid_start, yid_start)]), 1, xslice_type , neighbour[
    ↪ fr], tag, &(u[INDEX_2D_to_1D(xid_start, yid_end+1)]),1,xslice_type,neighbour[
    ↪ be],tag,mycomm,&statut);
MPI_Sendrecv(&(u[INDEX_2D_to_1D(xid_start, yid_end)]), 1, xslice_type, neighbour[be
    ↪ ], tag, &(u[INDEX_2D_to_1D(xid_start, yid_start-1)]),1,xslice_type, neighbour
    ↪ [fr], tag, mycomm, &statut);
```

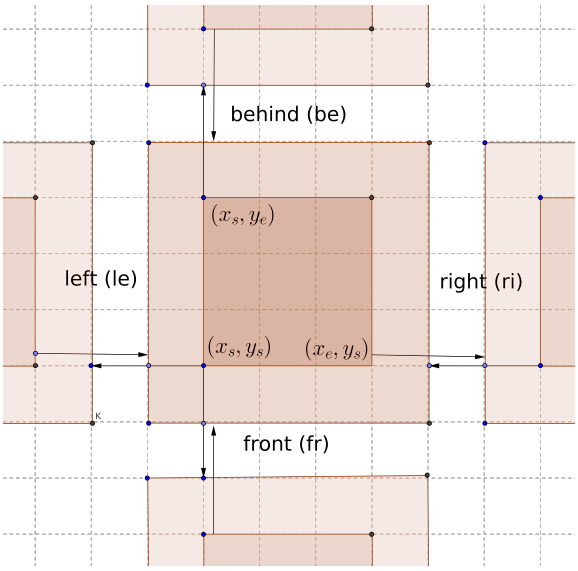


Figure 5: The communication between a process (center) and its neighbours in 2D.

Similarly, the communication in 3D will be done by the following code

```
// exchange data in yz-plane
MPI_Sendrecv(&(p0[INDEX_3D_to_1D(xid_start, yid_start,zid_start)]), 1, yzslice_type,
    ↪  neighbour[le], tag, &(p0[INDEX_3D_to_1D(xid_end+1, yid_start,zid_start)]),
    ↪ 1, yzslice_type, neighbour[ri], tag, mycomm, &statut);
MPI_Sendrecv(&(p0[INDEX_3D_to_1D(xid_end,yid_start,zid_start)]), 1, yzslice_type,
    ↪ neighbour[ri], tag,&(p0[INDEX_3D_to_1D(xid_start-1, yid_start,zid_start)]),
    ↪ 1, yzslice_type, neighbour[le], tag, mycomm, &statut);
// exchange data in zx-plane
MPI_Sendrecv(&(p0[INDEX_3D_to_1D(xid_start, yid_start,zid_start)]), 1, zxslice_type
    ↪ , neighbour[fr], tag, &(p0[INDEX_3D_to_1D(xid_start, yid_end+1,zid_start)])
    ↪ ,1,zxslice_type,neighbour[be],tag,mycomm,&statut);
MPI_Sendrecv(&(p0[INDEX_3D_to_1D(xid_start, yid_end, zid_start)]), 1, zxslice_type,
    ↪ neighbour[be], tag, &(p0[INDEX_3D_to_1D(xid_start, yid_start-1,zid_start)])
    ↪ ,1,zxslice_type, neighbour[fr], tag, mycomm, &statut);
```

```
// exchange data in xy-plane
MPI_Sendrecv(&(p0[INDEX_3D_to_1D(xid_start, yid_start,zid_start)]), 1, xyslice_type
    ↪ , neighbour[bo], tag, &(p0[INDEX_3D_to_1D(xid_start, yid_start,zid_end+1)])
    ↪ ,1,xyslice_type,neighbour[to],tag,mycomm,&statut);
MPI_Sendrecv(&(p0[INDEX_3D_to_1D(xid_start, yid_start, zid_end)]), 1, xyslice_type,
    ↪ neighbour[to], tag, &(p0[INDEX_3D_to_1D(xid_start, yid_start,zid_start-1)])
    ↪ ,1,xyslice_type, neighbour[bo], tag, mycomm, &statut);
```

4 Measurements

To measure the accuracy of the solvers, we consider the maximum error of the absolute difference between the exact solution u_e and the numerical solution u, i.e

$$max\_error = \|u - u_e\|_\infty \approx \max_i |u_e^i - u^i| \tag{7}$$

where i indicates the grid point index.

We then introduce two quantities used to measure the efficiency of the MPI implementation: timing and speedup.

Timing is defined as the time elapsed from that instance at which the first processor starts program execution to that instance at which the last processor completes it. The timing in seconds is the difference between two time values that generated from two calls of `MPI_Wtime()`. Since the most expensive part of the program is the solver, we call `MPI_Wtime()` before and after calling the solver, i.e

```
double tstart = MPI_Wtime();
cg_mpi(u, &it, eps,MAX_ITERATION);
double tend = MPI_Wtime();
double Timing = tend-tstart;
```

Speedup [6] is the ratio between timing for serial execution T_{serial} and timing for parallel execution $T_{parallel}$, i.e

$$S = \frac{T_{serial}}{T_{parallel}} \tag{8}$$

The ideal speedup is $S_{ideal}(p) = p$, i.e when p MPI processes are used, the parallel execution will be p times faster than the serial execution.

Since the timing of one program may be different for different executions, it is more reliable to execute a program many times and measure its mean [3] and standard deviation [7]. Assuming that we execute one program M times with corresponding timing of $T_1, T_2, \ldots, T_M$, the mean is then computed by Eq. 9

$$\bar{T} = \frac{1}{M}\sum_{i=1}^{M} T_i \tag{9}$$

To measure how well the mean represents for the data, one uses the standard deviation Eq. 10

$$\sigma_T = \sqrt{\frac{1}{M-1}\sum_{i=1}^{M}(T_i - \bar{T})^2} \tag{10}$$

5 Results

This report is enclosed with the implementation of parallel version of CG method in C for both 2D and 3D. Readers may want to look at appendix A to see how to perform the computation.

All computations were performed on the PDC's supercomputer Beskow `https://www.pdc.kth.se/resources/computers/beskow` using 2 nodes with 64 MPI processes in total, i.e 32 MPI processes for each node. For simplification, we use N to represent the size of linear systems, i.e N=NX=NY for 2D and N=NX=NY=NZ for 3D although NX, NY and NZ are not necessarily equal to make our code to work. The maximum number of iterations is set to `MAX_ITERATION=50000` and the error tolerance is `eps=1e-4`.

We start with a 2D problem on $\Omega = [-1, 1]^2$. The source is $f(x, y) = 40\,(-1+x^2+y^2)\exp(-x^2 - y^2)$ corresponding to the exact solution $u_e(x, y) = 10\exp(-x^2 - y^2)$ (Fig. 6). The number of iterations needed for the convergence and the maximum error computed by Eq. 7 is provided in table 3.

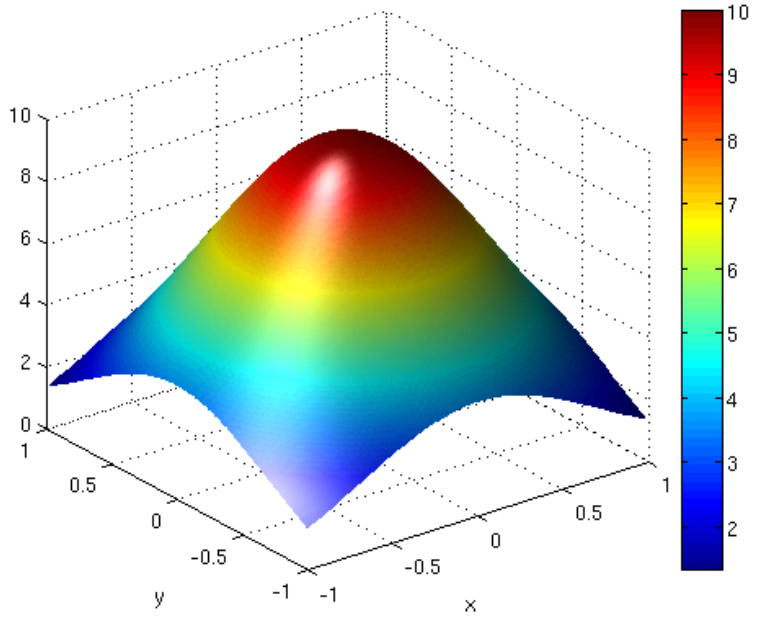


Figure 6: The exact solution $u_e(x, y) = 10\exp(-x^2 - y^2)$ corresponding to the source $f(x, y) = 40\,(-1 + x^2 + y^2)\exp(-x^2 - y^2)$.

We first study the affect of space discretization on the timing and speedup by considering five system sizes, i.e N=256, 512, 1024, 2048 and 4096. To have more reliable analysis, we run 10 times for each system size and the mean value of timing is used to analyze the performance. The data is provided in appendix B. Since the standard deviation is rather small, the results are very similar for different times of running.

For smaller-scale systems (N=256, 512), the timing increases and the speedup drops when more

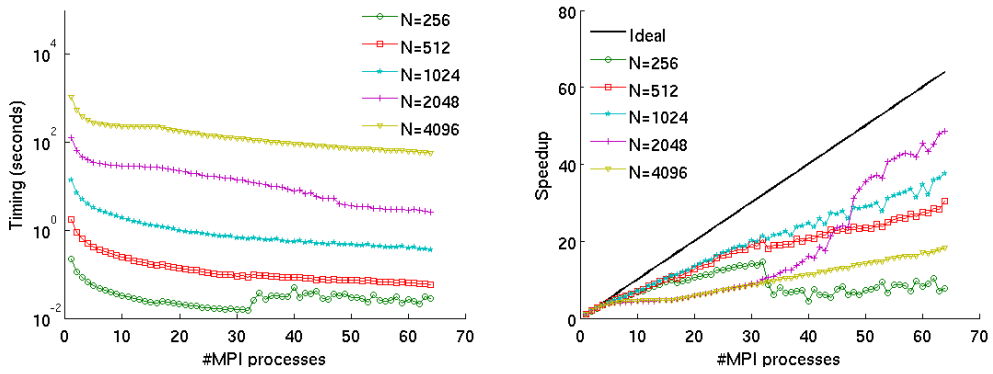
than 32 MPI processes are used (see Fig. 7a and Fig. 7b). It reflects that the communication of MPI processes between two nodes takes longer than that within one node. However, this drop is neglectable for larger-scale systems (N=1024, 2048, 4096) since the computational time within each MPI process compensates for the communication time.

For N=2048 and 4096, the speedup slows down and remains nearly unchanged when more than 5 processes are used. However, going beyond 15 processes, we regain speedup. A possible reason may be relevant to the unbalanced distribution of processes per coordinate direction using `MPI_Dims_create` that is listed in table 1. The speedup curve drops at some points where the number of processes is prime, for instance, #procs=47, 53, 59, 61 (see Fig. 7b). It is reasonable since the data is only parallelized in $x-$direction.

$1 = 1 \times 1$	$2 = 2 \times 1$	$3 = 3 \times 1$	$4 = 2 \times 2$	$5 = 5 \times 1$	$6 = 3 \times 2$	$7 = 7 \times 1$	$8 = 4 \times 2$
$9 = 3 \times 3$	$10 = 5 \times 2$	$11 = 11 \times 1$	$12 = 4 \times 3$	$13 = 13 \times 1$	$14 = 7 \times 2$	$15 = 5 \times 3$	$16 = 4 \times 4$
$17 = 17 \times 1$	$18 = 6 \times 3$	$19 = 19 \times 1$	$20 = 5 \times 4$	$21 = 7 \times 3$	$22 = 11 \times 2$	$23 = 23 \times 1$	$24 = 6 \times 4$
$25 = 5 \times 5$	$26 = 13 \times 2$	$27 = 9 \times 3$	$28 = 7 \times 4$	$29 = 29 \times 1$	$30 = 6 \times 5$	$31 = 31 \times 1$	$32 = 8 \times 4$
$33 = 11 \times 3$	$34 = 17 \times 2$	$35 = 7 \times 5$	$36 = 6 \times 6$	$37 = 37 \times 1$	$38 = 19 \times 2$	$39 = 13 \times 3$	$40 = 8 \times 5$
$41 = 41 \times 1$	$42 = 7 \times 6$	$43 = 43 \times 1$	$44 = 11 \times 4$	$45 = 9 \times 5$	$46 = 23 \times 2$	$47 = 47 \times 1$	$48 = 8 \times 6$
$49 = 7 \times 7$	$50 = 10 \times 5$	$51 = 17 \times 3$	$52 = 13 \times 4$	$53 = 53 \times 1$	$54 = 9 \times 6$	$55 = 11 \times 5$	$56 = 8 \times 7$
$57 = 19 \times 3$	$58 = 29 \times 2$	$59 = 59 \times 1$	$60 = 10 \times 6$	$61 = 61 \times 1$	$62 = 31 \times 2$	$63 = 9 \times 7$	$64 = 8 \times 8$

Table 1: Distribution of MPI processes versus directions automatically generated by `MPI_Dims_create` for 2D.

To summarize, we can say that for smaller-scale systems, speedup is efficient if we have MPI processes within the limits of number of processors/node. For larger systems going beyond 1 node, communication penalty is not so high as in the case of smaller systems, so we still have speedup. However, the speedup gain corresponding to the number of MPI processes is debatable.

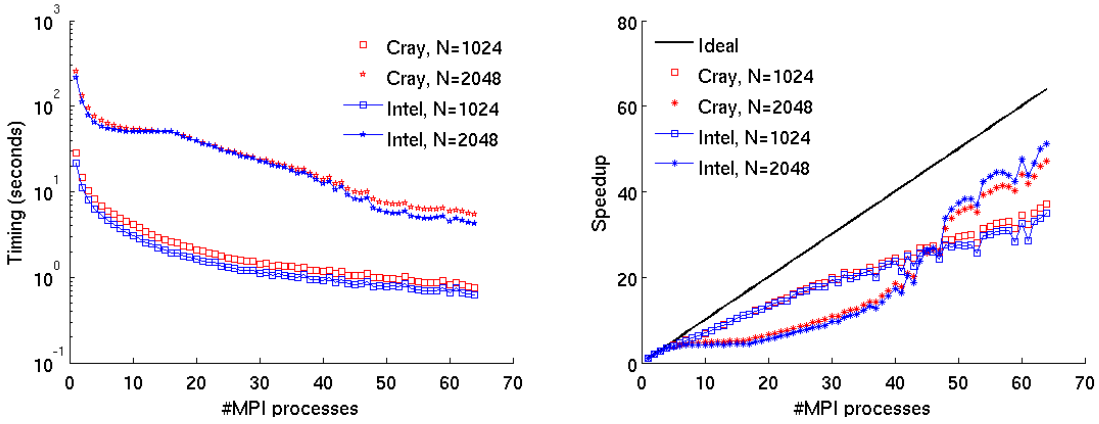


(a) (b)

Figure 7: Timing and speedup for 2D with Cray compiler for CG with `eps=1e-4` and six system sizes, i.e N=128, 256, 512, 1024, 2048 and 4096.

Now we test the performance on different compilers, i.e Cray and Intel for two system sizes,

N=1024 and N=2048. We note that, since the standard deviation is rather small, we did not run many times for each system size on Intel. We pick up the results from the first trial. In general, the computation on Intel is faster (Fig. 8a). However, the speedup is similar for both cases (Fig. 8b).



(a) (b)

Figure 8: Timing and speedup for CG in 2D on Cray and Intel with `eps=1e-4`, N=1024 and 2048. In general, the computation on Intel is faster. However, the speedup is similar for both cases.

For 3D, we chose the source $f(x, y, z) = 20\,(-3 + 2x^2 + 2y^2 + 2z^2)\exp(-x^2 - y^2 - z^2)$ corresponding to the exact solution $u_e(x, y, z) = 10\exp(-x^2 - y^2 - z^2)$ on $\Omega = [-1, 1]^3$ (Fig. 9). We also vary system sizes, i.e N=128, 256, 512 and 1024 to study the affect of space discretization. The number of iterations needed for the convergence and the maximum error computed by Eq. 7 is provided in table 8. Similarily to 2D, the timing of 10 times of executing (trials) for N=128 and 256 is not much different, i.e with small deviation (Tables 9 and 10).

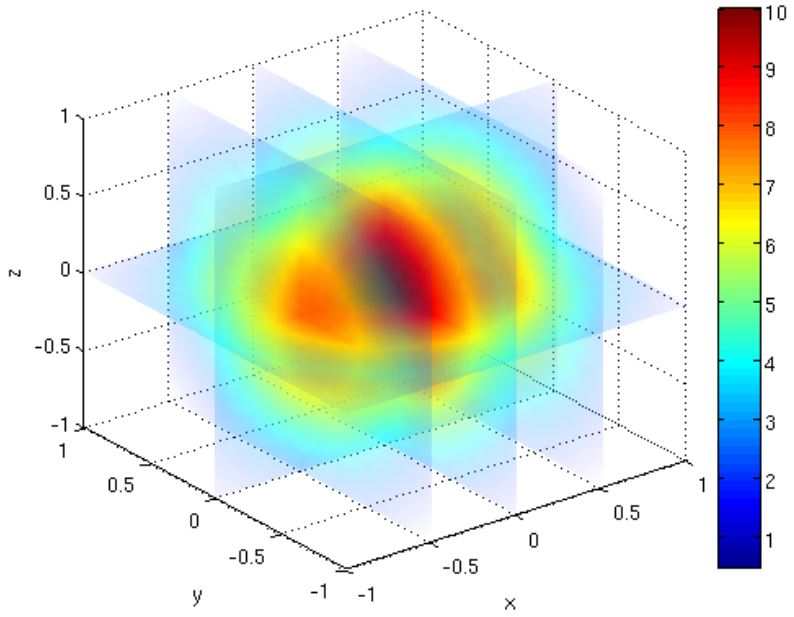


Figure 9: The exact solution $u_e(x, y, z) = 10 \exp(-x^2 - y^2 - z^2)$ corresponding to the source $f(x, y, z) = 20\,(-3 + 2x^2 + 2y^2 + 2z^2) \exp(-x^2 - y^2 - z^2)$.

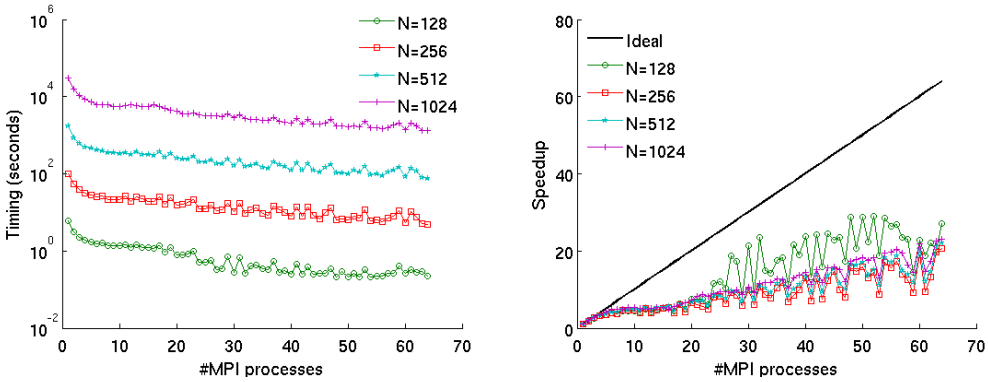
We first look at the distribution of MPI processes versus space directions. We denote P the number of MPI processes, P_x, P_y and P_z indicate the number of MPI processes distributed in $x-$, $y-$ and $z-$directions respectively, i.e $P = P_x \times P_y \times P_z$. For a problem size of $N \times N \times N$, an ideally balanced distribution requires $P_x = P_y = P_z$. The smaller the difference between P_x, P_y and P_z is, the better performance is.

Table 2 shows the distribution of MPI processes versus directions automatically generated by `MPI_Dims_create` in 3D. We can divide the data in the table into three groups corresponding to black, blue and red. In the red group, the number of MPI processes is prime. It consists of the most unbalanced distribution since the distribution is similar to 1D problem, i.e the data is only parallelized in the $x-$direction. The blue group has better distribution and it is similar to 2D problem, i.e the data is parallelized in the $x-$, and $y-$directions. When the number of MPI processes in black group is used, we obtain good performance since the data is parallelized in three directions. Since three groups are mixedly used, the computational time and speedup curves are oscillatory (Fig. 10), especially for large number of processes. However, when system size N is large enough compared to P_x, P_y and P_z, the speedup curve is smoother because the computational time can compensate for the communication.

In short, to obtain a good performance, it requires to choose a suitable number of MPI processes such that P_x, P_y, P_z are close together.

$1 = 1 \times 1 \times 1$	$2 = 2 \times 1 \times 1$	$3 = 3 \times 1 \times 1$	$4 = 2 \times 2 \times 1$
$5 = 5 \times 1 \times 1$	$6 = 3 \times 2 \times 1$	$7 = 7 \times 1 \times 1$	$8 = 2 \times 2 \times 2$
$9 = 3 \times 3 \times 1$	$10 = 5 \times 2 \times 1$	$11 = 11 \times 1 \times 1$	$12 = 3 \times 2 \times 2$
$13 = 13 \times 1 \times 1$	$14 = 7 \times 2 \times 1$	$15 = 5 \times 3 \times 1$	$16 = 4 \times 4 \times 1$
$17 = 17 \times 1 \times 1$	$18 = 3 \times 3 \times 2$	$19 = 19 \times 1 \times 1$	$20 = 5 \times 2 \times 2$
$21 = 7 \times 3 \times 1$	$22 = 11 \times 2 \times 1$	$23 = 23 \times 1 \times 1$	$24 = 4 \times 3 \times 2$
$25 = 5 \times 5 \times 1$	$26 = 13 \times 2 \times 1$	$27 = 3 \times 3 \times 3$	$28 = 7 \times 2 \times 2$
$29 = 29 \times 1 \times 1$	$30 = 5 \times 3 \times 2$	$31 = 31 \times 1 \times 1$	$32 = 4 \times 4 \times 2$
$33 = 11 \times 3 \times 1$	$34 = 17 \times 2 \times 1$	$35 = 7 \times 5 \times 1$	$36 = 4 \times 3 \times 3$
$37 = 37 \times 1 \times 1$	$38 = 19 \times 2 \times 1$	$39 = 13 \times 3 \times 1$	$40 = 5 \times 4 \times 2$
$41 = 41 \times 1 \times 1$	$42 = 7 \times 3 \times 2$	$43 = 43 \times 1 \times 1$	$44 = 11 \times 2 \times 2$
$45 = 5 \times 3 \times 3$	$46 = 23 \times 2 \times 1$	$47 = 47 \times 1 \times 1$	$48 = 4 \times 4 \times 3$
$49 = 7 \times 7 \times 1$	$50 = 5 \times 5 \times 2$	$51 = 17 \times 3 \times 1$	$52 = 13 \times 2 \times 2$
$53 = 53 \times 1 \times 1$	$54 = 6 \times 3 \times 3$	$55 = 11 \times 5 \times 1$	$56 = 7 \times 4 \times 2$
$57 = 19 \times 3 \times 1$	$58 = 29 \times 2 \times 1$	$59 = 59 \times 1 \times 1$	$60 = 5 \times 4 \times 3$
$61 = 61 \times 1 \times 1$	$62 = 31 \times 2 \times 1$	$63 = 7 \times 3 \times 3$	$64 = 4 \times 4 \times 4$

Table 2: Distribution of MPI processes versus directions automatically generated by `MPI_Dims_create` ↪ . There are three groups in the table corresponding to black, blue and red. In the red group, the number of MPI processes is prime. It consists of the most unbalanced distribution since the distribution is similar to 1D problem, i.e the data is only parallelized in the $x-$direction. The blue group has better distribution and it is similar to 2D problem, i.e the data is parallelized in the $x-$, and $y-$directions. We will obtain good performance when using number of processes in black group since the data is parallelized in three directions, i.e P_x, P_y and P_z are quite close.



(a) (b)

Figure 10: Timing and speedup for CG in 3D with Cray compiler, `eps=1e-4` and four system sizes, i.e N=128, 256, 512 and 1024.

6 Conclusions

In this project, we developed an MPI-based conjugate gradient method for both 2D and 3D. The performance of MPI is affected by many factors. To obtain a good performance, we need to adjust the relationship between space discretization with the number of processes. For a problem size of N for all dimensional directions, an ideally balanced distribution requires that the number of MPI processes on each dimensional direction is close together. The number of MPI processes per node should be noticed as well.

Appendices

A How to run the code

In Beskow, the program needs to be compiled with `cc` and executed with `aprun`. The input consists of the solver name, `J` for Jacobi and `C` for conjugate gradient, the error tolerance `eps`, the maximum number of iterations `MAX_ITERATION` and the space discretization, i.e NX, NY and NZ. We have to note that, however, the Jacobi method [5] has a slow convergence and is not analyzed in this report.

For 2D, to solve the equation by CG (`C`) method with `eps=1e-6`, `MAX_ITERATION=10000` and `N=256`, the batch file for 2D, `run2d.sh`, can be the following

```
cc poisson2d.c -o poisson2d -lm
for n in $(seq 1 64); do
    export MPI_NUM_THREADS=$n
    aprun -n $MPI_NUM_THREADS ./poisson2d C 1e-6 10000 256 256
done
```

It is similar for 3D except that we need to specify `NZ`. In `run3d.sh`, we can set

```
cc poisson3d.c -o poisson3d -lm
for n in $(seq 1 64); do
    export MPI_NUM_THREADS=$n
    aprun -n $MPI_NUM_THREADS ./poisson3d C 1e-4 50000 256 256 256
done
```

B Number of iterations, accurracy and timing of using CG for 2D on Cray compiler

System size N	# iterations	max error
256	499	1.6e-4
512	921	3.9e-5
1024	1890	9.9e-6
2048	3904	2.5e-6
4096	8006	6.2e-7

Table 3: Number of iterations and maximum error of CG for 2D problem with `eps=1e-4` and different system sizes.

#procs	Trial										$\bar{T}$	σ_T
	1	2	3	4	5	6	7	8	9	10		
1	1.7	1.7	1.7	1.7	1.7	1.7	1.7	1.7	1.7	1.7	1.7	1.0e-3
2	9.1e-1	9.0e-1	8.9e-1	9.1e-1	9.0e-1	9.0e-1	8.9e-1	9.0e-1	9.0e-1	9.0e-1	9.0e-1	4.2e-3
3	6.5e-1	6.5e-1	6.5e-1	6.5e-1	6.4e-1	6.4e-1	6.5e-1	6.4e-1	6.4e-1	6.4e-1	6.4e-1	2.3e-3
4	5.0e-1	5.0e-1	5.0e-1	5.0e-1	5.0e-1	5.0e-1	5.0e-1	5.0e-1	5.0e-1	5.0e-1	5.0e-1	2.9e-3
5	4.2e-1	4.2e-1	4.2e-1	4.2e-1	4.2e-1	4.2e-1	4.2e-1	4.2e-1	4.2e-1	4.2e-1	4.2e-1	1.9e-3
6	3.6e-1	3.7e-1	3.6e-1	3.6e-1	3.6e-1	3.6e-1	3.6e-1	3.6e-1	3.6e-1	3.6e-1	3.6e-1	2.4e-3
7	3.3e-1	3.2e-1	3.3e-1	3.3e-1	3.3e-1	3.3e-1	3.2e-1	3.2e-1	3.3e-1	3.3e-1	3.3e-1	2.2e-3
8	3.0e-1	3.0e-1	3.0e-1	3.0e-1	3.0e-1	3.0e-1	3.0e-1	3.0e-1	3.0e-1	3.0e-1	3.0e-1	9.5e-4
9	2.7e-1	2.7e-1	2.7e-1	2.7e-1	2.7e-1	2.7e-1	2.7e-1	2.7e-1	2.7e-1	2.7e-1	2.7e-1	5.2e-4
10	2.5e-1	2.5e-1	2.5e-1	2.5e-1	2.5e-1	2.5e-1	2.5e-1	2.5e-1	2.5e-1	2.5e-1	2.5e-1	5.2e-4
11	2.3e-1	2.3e-1	2.3e-1	2.3e-1	2.3e-1	2.3e-1	2.3e-1	2.3e-1	2.3e-1	2.3e-1	2.3e-1	5.2e-4
12	2.1e-1	2.1e-1	2.1e-1	2.1e-1	2.1e-1	2.1e-1	2.1e-1	2.1e-1	2.1e-1	2.1e-1	2.1e-1	6.1e-4
13	2.0e-1	2.0e-1	2.0e-1	2.0e-1	2.0e-1	2.0e-1	2.0e-1	2.0e-1	2.0e-1	2.0e-1	2.0e-1	6.8e-4
14	1.8e-1	1.8e-1	1.8e-1	1.8e-1	1.8e-1	1.8e-1	1.8e-1	1.8e-1	1.8e-1	1.8e-1	1.8e-1	3.2e-4
15	1.7e-1	1.7e-1	1.7e-1	1.7e-1	1.7e-1	1.7e-1	1.7e-1	1.7e-1	1.7e-1	1.7e-1	1.7e-1	5.8e-4
16	1.6e-1	1.6e-1	1.6e-1	1.6e-1	1.6e-1	1.6e-1	1.6e-1	1.6e-1	1.6e-1	1.6e-1	1.6e-1	4.1e-4
48	7.3e-2	7.3e-2	7.3e-2	7.3e-2	7.3e-2	7.3e-2	7.3e-2	7.3e-2	7.3e-2	7.3e-2	7.3e-2	1.5e-4
49	7.4e-2	7.6e-2	7.5e-2	7.4e-2	7.5e-2	7.4e-2	7.6e-2	7.4e-2	7.6e-2	7.4e-2	7.5e-2	8.4e-4
50	7.5e-2	7.4e-2	7.3e-2	7.4e-2	7.5e-2	7.4e-2	7.4e-2	7.4e-2	7.4e-2	7.5e-2	7.4e-2	5.4e-4
51	7.5e-2	7.4e-2	7.4e-2	7.5e-2	7.4e-2	7.4e-2	7.5e-2	7.5e-2	7.4e-2	7.4e-2	7.5e-2	4.6e-4
52	7.0e-2	7.1e-2	7.1e-2	7.1e-2	7.1e-2	7.1e-2	7.2e-2	7.1e-2	7.1e-2	7.1e-2	7.1e-2	4.4e-4
53	7.4e-2	7.4e-2	7.4e-2	7.4e-2	7.3e-2	7.3e-2	7.3e-2	7.4e-2	7.3e-2	7.4e-2	7.4e-2	4.0e-4
54	6.9e-2	7.0e-2	7.0e-2	7.0e-2	6.9e-2	6.9e-2	7.0e-2	6.9e-2	7.0e-2	7.0e-2	7.0e-2	3.6e-4
55	6.8e-2	6.8e-2	6.8e-2	6.8e-2	6.8e-2	6.8e-2	6.8e-2	6.8e-2	6.9e-2	6.8e-2	6.8e-2	2.9e-4
56	6.7e-2	6.6e-2	6.6e-2	6.6e-2	6.7e-2	6.6e-2	6.7e-2	6.7e-2	6.6e-2	6.6e-2	6.7e-2	2.0e-4
57	6.7e-2	6.7e-2	6.7e-2	6.7e-2	6.7e-2	6.7e-2	6.7e-2	6.7e-2	6.7e-2	6.7e-2	6.7e-2	1.5e-4
58	6.4e-2	6.4e-2	6.4e-2	6.4e-2	6.4e-2	6.4e-2	6.4e-2	6.5e-2	6.4e-2	6.4e-2	6.4e-2	1.8e-4
59	6.6e-2	6.6e-2	6.6e-2	6.6e-2	6.6e-2	6.6e-2	6.6e-2	6.6e-2	6.6e-2	6.6e-2	6.6e-2	1.6e-4
60	6.3e-2	6.3e-2	6.3e-2	6.3e-2	6.3e-2	6.3e-2	6.4e-2	6.3e-2	6.3e-2	6.3e-2	6.3e-2	2.3e-4
61	6.4e-2	6.4e-2	6.4e-2	6.4e-2	6.4e-2	6.4e-2	6.4e-2	6.4e-2	6.4e-2	6.4e-2	6.4e-2	1.1e-4
62	6.1e-2	6.1e-2	6.1e-2	6.1e-2	6.1e-2	6.1e-2	6.1e-2	6.1e-2	6.2e-2	6.1e-2	6.1e-2	2.0e-4
63	6.2e-2	6.2e-2	6.1e-2	6.2e-2	6.2e-2	6.1e-2	6.1e-2	6.2e-2	6.2e-2	6.2e-2	6.2e-2	2.3e-4
64	5.7e-2	5.7e-2	5.8e-2	5.7e-2	5.7e-2	5.7e-2	5.7e-2	5.7e-2	5.7e-2	5.7e-2	5.7e-2	2.0e-4

Table 4: Timing (in seconds) for 10 times of executing (trials) and its mean $\bar{T}$, standard deviation σ_T for 2D with N=512. We varied the number of MPI processes from 1 to 64.

#procs	Trial										$\bar{T}$	σ_T
	1	2	3	4	5	6	7	8	9	10		
1	13.63	13.64	13.69	13.65	13.61	13.63	13.67	13.64	13.67	13.65	13.65	2.4e-2
2	7.08	7.11	7.06	7.09	7.08	7.01	6.98	7.06	7.06	7.01	7.05	4.2e-2
3	5.00	5.03	5.05	5.01	5.02	5.07	5.04	5.01	5.03	5.02	5.03	2.1e-2
4	3.98	3.92	3.95	3.93	3.95	3.95	3.99	3.95	3.98	3.95	3.95	2.3e-2
5	3.32	3.36	3.33	3.40	3.31	3.36	3.34	3.31	3.37	3.35	3.34	2.7e-2
6	2.89	2.89	2.90	2.90	2.92	2.89	2.88	2.93	2.89	2.89	2.90	1.6e-2
7	2.64	2.61	2.62	2.63	2.60	2.64	2.60	2.61	2.62	2.62	2.62	1.4e-2
8	2.41	2.41	2.40	2.39	2.40	2.37	2.37	2.39	2.37	2.38	2.39	1.5e-2
9	2.15	2.18	2.15	2.15	2.15	2.17	2.15	2.15	2.18	2.15	2.16	1.4e-2
10	1.94	1.97	1.97	1.93	1.96	1.94	1.95	1.96	1.94	1.94	1.95	1.3e-2
11	1.81	1.78	1.78	1.81	1.78	1.78	1.79	1.79	1.80	1.80	1.79	1.2e-2
12	1.63	1.64	1.62	1.62	1.65	1.65	1.64	1.64	1.62	1.62	1.63	1.3e-2
13	1.52	1.52	1.52	1.53	1.52	1.52	1.52	1.52	1.56	1.55	1.53	1.4e-2
14	1.40	1.43	1.43	1.41	1.42	1.40	1.40	1.43	1.41	1.41	1.41	1.2e-2
15	1.33	1.31	1.31	1.34	1.34	1.33	1.31	1.31	1.31	1.32	1.32	1.0e-2
16	1.23	1.25	1.23	1.23	1.25	1.23	1.25	1.25	1.25	1.23	1.24	1.1e-2
						.....						
48	0.48	0.47	0.48	0.47	0.47	0.48	0.48	0.48	0.48	0.48	0.48	2.8e-3
49	0.47	0.47	0.47	0.47	0.48	0.48	0.48	0.48	0.48	0.49	0.48	6.6e-3
50	0.47	0.47	0.47	0.46	0.47	0.46	0.48	0.47	0.47	0.48	0.47	6.8e-3
51	0.46	0.46	0.46	0.45	0.47	0.47	0.47	0.47	0.47	0.48	0.47	6.7e-3
52	0.46	0.45	0.46	0.45	0.46	0.45	0.46	0.46	0.46	0.46	0.46	3.4e-3
53	0.48	0.49	0.49	0.49	0.49	0.48	0.49	0.50	0.50	0.49	0.49	4.6e-3
54	0.44	0.45	0.44	0.44	0.43	0.44	0.44	0.43	0.44	0.44	0.44	5.5e-3
55	0.43	0.42	0.43	0.43	0.42	0.43	0.43	0.42	0.43	0.43	0.43	4.3e-3
56	0.42	0.42	0.42	0.42	0.42	0.42	0.42	0.42	0.42	0.42	0.42	1.8e-3
57	0.42	0.42	0.41	0.42	0.42	0.42	0.42	0.41	0.42	0.42	0.42	4.6e-3
58	0.41	0.41	0.40	0.41	0.41	0.40	0.41	0.40	0.41	0.41	0.41	3.6e-3
59	0.44	0.43	0.43	0.44	0.43	0.43	0.44	0.44	0.44	0.44	0.43	4.6e-3
60	0.40	0.39	0.39	0.39	0.39	0.39	0.40	0.39	0.40	0.39	0.39	5.2e-3
61	0.43	0.42	0.43	0.43	0.42	0.42	0.44	0.43	0.43	0.42	0.43	7.4e-3
62	0.38	0.38	0.39	0.39	0.38	0.38	0.38	0.38	0.38	0.38	0.38	4.7e-3
63	0.37	0.37	0.38	0.38	0.38	0.38	0.37	0.38	0.37	0.37	0.37	4.5e-3
64	0.36	0.36	0.37	0.36	0.36	0.36	0.36	0.36	0.36	0.37	0.36	4.2e-3

Table 5: Timing (in seconds) for 10 times of executing (trials) and its mean $\bar{T}$, standard deviation σ_T for 2D with N=1024. We varied the number of MPI processes from 1 to 64.

#procs	Trial										$\bar{T}$	σ_T
	1	2	3	4	5	6	7	8	9	10		
1	127.8	127.1	127.4	127.4	127.5	127.1	127.4	127.4	127.4	128.1	127.5	3.1e-1
2	66.05	65.80	65.84	66.03	65.91	66.30	66.18	65.63	65.98	65.87	65.96	1.9e-1
3	47.25	47.50	47.07	47.38	47.40	47.34	47.46	47.23	47.63	47.59	47.39	1.7e-1
4	39.11	39.64	39.49	39.17	39.15	39.08	38.85	39.84	39.34	39.45	39.31	3.0e-1
5	35.75	35.54	35.41	35.28	35.09	35.22	35.51	35.32	35.50	35.61	35.42	2.0e-1
6	33.58	33.55	33.69	33.17	33.44	33.12	33.01	33.41	32.81	33.04	33.28	2.9e-1
7	31.65	31.89	31.41	31.70	31.37	31.58	31.64	31.61	31.36	31.60	31.58	1.6e-1
8	31.28	31.16	31.26	30.55	30.46	30.36	30.78	30.87	30.71	30.41	30.78	3.5e-1
9	29.69	30.02	29.52	29.78	29.65	30.13	29.64	29.67	29.37	29.65	29.71	2.2e-1
10	30.07	28.98	29.07	29.38	28.82	29.01	29.31	28.99	28.90	29.08	29.16	3.6e-1
11	29.18	29.68	29.15	29.08	29.08	28.85	29.17	28.97	29.18	29.16	29.15	2.1e-1
12	29.46	28.32	28.85	28.18	28.06	28.12	28.49	28.28	28.29	28.06	28.41	4.4e-1
13	28.25	28.63	28.31	28.50	28.31	28.84	28.25	28.26	28.40	28.36	28.41	1.9e-1
14	28.37	27.88	27.70	27.68	27.77	27.80	27.95	27.92	27.78	27.86	27.87	2.0e-1
15	27.76	27.63	28.34	27.97	27.61	27.49	27.64	27.44	27.61	27.85	27.73	2.7e-1
16	28.24	27.76	27.51	27.57	27.46	27.70	28.29	27.95	27.67	27.57	27.77	3.0e-1
						.....						
48	4.13	4.06	4.07	4.09	4.07	4.07	4.12	4.07	4.03	4.07	4.08	2.9e-2
49	3.89	3.78	3.77	3.77	3.80	3.84	3.75	3.76	3.81	3.77	3.80	4.3e-2
50	3.66	3.59	3.61	3.59	3.61	3.57	3.60	3.62	3.59	3.59	3.60	2.4e-2
51	3.48	3.48	3.47	3.49	3.48	3.49	3.47	3.48	3.50	3.46	3.48	1.2e-2
52	3.44	3.44	3.43	3.43	3.44	3.44	3.41	3.45	3.44	3.43	3.43	9.5e-3
53	3.49	3.50	3.52	3.51	3.50	3.50	3.50	3.52	3.49	3.50	3.50	1.0e-2
54	3.14	3.14	3.14	3.14	3.13	3.16	3.14	3.13	3.13	3.14	3.14	9.5e-3
55	3.08	3.08	3.08	3.08	3.07	3.08	3.07	3.08	3.07	3.07	3.08	5.0e-3
56	3.01	3.01	3.02	3.01	3.01	3.01	3.01	3.01	3.01	3.00	3.01	4.2e-3
57	2.98	2.98	2.98	2.97	2.97	2.98	2.97	2.99	2.98	2.98	2.98	5.2e-3
58	2.99	2.98	2.99	2.99	2.98	2.99	2.99	2.99	2.98	2.99	2.99	3.6e-3
59	3.03	3.03	3.04	3.03	3.04	3.03	3.04	3.04	3.04	3.03	3.03	5.2e-3
60	2.80	2.79	2.80	2.80	2.79	2.80	2.80	2.80	2.80	2.80	2.80	2.9e-3
61	2.94	2.95	2.95	2.95	2.94	2.94	2.94	2.94	2.94	2.94	2.94	4.0e-3
62	2.81	2.81	2.81	2.81	2.81	2.81	2.81	2.81	2.81	2.81	2.81	1.6e-3
63	2.67	2.67	2.67	2.67	2.67	2.67	2.66	2.67	2.67	2.67	2.67	2.3e-3
64	2.62	2.62	2.62	2.62	2.62	2.62	2.62	2.62	2.62	2.62	2.62	1.8e-3

Table 6: Timing (in seconds) for 10 times of executing (trials) and its mean $\bar{T}$, standard deviation σ_T for 2D with N=2048. We varied the number of MPI processes from 1 to 64.

#procs	Trial										$\bar{T}$	σ_T
	1	2	3	4	5	6	7	8	9	10		
1	1045.2	1043.2	1044.3	1043.6	1047.2	1043.1	1045.2	1044.2	1044.1	1042.6	1044.3	1.3
2	533.80	534.11	533.01	537.54	538.96	532.41	535.27	535.65	535.54	535.57	535.19	2.0
3	382.72	385.22	384.92	390.90	393.22	383.25	386.49	384.29	383.95	382.56	385.75	3.6
4	315.56	317.48	316.33	313.08	319.19	318.04	315.31	315.19	317.17	320.87	316.82	2.2
5	281.96	286.40	281.08	281.54	284.31	284.22	283.97	279.29	280.89	277.73	282.14	2.6
6	261.99	261.82	262.74	261.53	261.15	261.88	260.87	259.32	261.46	260.39	261.32	9.5e-1
7	249.44	248.90	251.36	247.75	248.36	250.53	250.98	252.39	250.63	250.30	250.06	1.4
8	241.28	241.67	242.95	241.94	242.05	240.09	241.21	241.58	241.44	241.60	241.58	7.2e-1
9	235.53	235.57	235.13	235.52	235.92	234.31	235.79	239.22	233.41	235.50	235.59	1.5
10	230.95	230.85	231.07	231.94	233.15	231.81	233.44	231.43	230.43	231.42	231.65	9.8e-1
11	229.52	231.24	229.68	229.88	229.32	229.84	229.42	228.73	228.27	227.49	229.34	1.0
12	225.16	226.65	225.71	226.58	227.46	224.61	225.31	224.25	226.88	225.39	225.80	1.0
13	226.82	228.46	227.18	227.09	227.37	225.98	226.43	226.41	227.69	227.71	227.11	7.4e-1
14	224.08	226.47	224.09	224.38	225.74	224.06	225.36	224.96	225.31	224.19	224.86	8.4e-1
15	224.44	223.70	223.50	224.22	225.80	224.40	226.16	224.11	224.10	223.91	224.44	8.7e-1
16	224.31	223.47	225.00	224.33	223.86	224.59	224.92	223.47	224.68	225.22	224.39	6.2e-1
						.....						
48	76.21	76.06	75.85	76.12	76.05	76.44	76.26	76.91	75.77	76.39	76.21	3.3e-1
49	74.48	74.38	75.04	74.88	74.95	75.06	74.76	75.36	74.91	75.27	74.91	3.1e-1
50	72.93	73.81	73.11	73.11	73.02	73.49	73.16	73.35	73.32	73.64	73.29	2.8e-1
51	72.31	72.33	72.04	72.31	72.32	72.68	72.51	72.49	72.79	72.71	72.45	2.3e-1
52	71.30	71.23	71.30	71.23	71.36	71.67	71.81	71.57	71.44	71.70	71.46	2.1e-1
53	71.24	71.21	71.41	71.43	71.12	71.37	71.61	71.39	71.54	71.51	71.38	1.5e-1
54	67.84	67.99	67.70	67.94	67.51	68.10	67.83	68.05	67.87	67.56	67.84	2.0e-1
55	66.59	66.85	66.90	66.55	67.06	67.19	67.29	66.89	66.54	66.84	66.87	2.6e-1
56	65.79	65.49	65.50	65.51	65.31	65.35	65.98	65.61	66.05	65.36	65.59	2.6e-1
57	64.80	64.72	64.79	64.91	64.68	65.13	64.97	65.06	65.19	64.72	64.90	1.8e-1
58	64.60	64.05	64.00	64.49	64.59	64.28	64.26	64.35	64.71	64.42	64.38	2.3e-1
59	64.59	64.37	64.21	65.00	64.31	64.99	64.57	64.12	64.87	64.59	64.56	3.1e-1
60	61.18	61.36	61.22	60.96	61.06	62.12	61.67	61.76	61.76	61.45	61.45	3.7e-1
61	62.56	62.45	62.71	62.27	62.79	62.21	62.55	62.25	62.35	62.48	62.46	1.9e-1
62	60.52	60.13	60.28	59.93	59.98	60.21	60.29	60.54	60.67	61.07	60.36	3.5e-1
63	58.38	57.83	58.81	58.27	58.55	58.93	58.49	58.22	58.79	58.32	58.46	3.3e-1
64	57.30	57.11	56.98	57.25	57.06	58.45	57.85	57.44	57.64	57.55	57.46	4.4e-1

Table 7: Timing (in seconds) for 10 times of executing (trials) and its mean $\bar{T}$, standard deviation σ_T for 2D with N=4096. We varied the number of MPI processes from 1 to 64.

C Number of iterations, accurracy and timing of using CG for 3D on Cray compiler

System size N	# iterations	max error
128	295	6.9e-4
256	616	1.7e-4
512	1284	4.4e-5
1024	2680	1.1e-5

Table 8: Number of iterations and maximum error of CG for 3D problem with `eps=1e-4` and different system sizes.

#procs	Trial										$\bar{T}$	σ_T
	1	2	3	4	5	6	7	8	9	10		
1	6.04	6.05	5.99	6.00	6.04	6.01	5.97	6.04	6.01	5.97	6.01	2.9e-2
2	3.14	3.15	3.23	3.13	3.15	3.22	3.16	3.15	3.14	3.15	3.16	3.6e-2
3	2.31	2.31	2.29	2.33	2.35	2.31	2.35	2.30	2.35	2.29	2.32	2.3e-2
4	1.89	1.90	1.89	1.92	1.88	1.85	1.82	1.83	1.83	1.90	1.87	3.4e-2
5	1.74	1.75	1.69	1.72	1.73	1.73	1.72	1.73	1.74	1.69	1.73	2.0e-2
6	1.55	1.55	1.55	1.57	1.58	1.56	1.50	1.52	1.54	1.56	1.55	2.3e-2
7	1.59	1.58	1.56	1.55	1.56	1.56	1.59	1.57	1.62	1.58	1.58	2.0e-2
8	1.38	1.38	1.38	1.39	1.40	1.37	1.29	1.32	1.29	1.38	1.36	4.1e-2
9	1.36	1.35	1.35	1.31	1.34	1.31	1.35	1.35	1.35	1.30	1.34	2.2e-2
10	1.34	1.33	1.30	1.39	1.36	1.33	1.31	1.29	1.29	1.34	1.33	3.1e-2
11	1.45	1.46	1.45	1.41	1.42	1.40	1.45	1.45	1.44	1.44	1.44	1.9e-2
12	1.25	1.21	1.16	1.21	1.22	1.21	1.21	1.21	1.16	1.24	1.21	2.7e-2
13	1.42	1.42	1.43	1.40	1.39	1.39	1.38	1.43	1.43	1.38	1.40	2.1e-2
14	1.26	1.25	1.22	1.26	1.26	1.25	1.27	1.22	1.22	1.26	1.25	1.8e-2
15	1.20	1.19	1.19	1.16	1.20	1.16	1.19	1.20	1.19	1.16	1.19	1.8e-2
16	1.14	1.14	1.09	1.14	1.14	1.14	1.10	1.10	1.10	1.13	1.12	2.1e-2
						.....						
48	0.21	0.31	0.21	0.20	0.21	0.21	0.20	0.21	0.20	0.20	0.22	3.4e-2
49	0.29	0.27	0.21	0.21	0.21	0.21	0.21	0.21	0.21	0.21	0.22	3.1e-2
50	0.21	0.25	0.20	0.20	0.20	0.20	0.20	0.20	0.20	0.20	0.21	1.4e-2
51	0.27	0.25	0.22	0.23	0.22	0.22	0.22	0.22	0.22	0.23	0.23	1.7e-2
52	0.21	0.25	0.20	0.21	0.20	0.20	0.21	0.21	0.21	0.20	0.21	1.6e-2
53	0.32	0.32	0.29	0.32	0.29	0.29	0.29	0.29	0.30	0.31	0.30	1.4e-2
54	0.21	0.24	0.18	0.18	0.18	0.18	0.18	0.19	0.18	0.19	0.19	1.8e-2
55	0.23	0.25	0.20	0.20	0.19	0.19	0.19	0.20	0.20	0.19	0.20	1.8e-2
56	0.22	0.26	0.18	0.18	0.18	0.18	0.18	0.18	0.18	0.18	0.19	2.7e-2
57	0.26	0.25	0.21	0.20	0.20	0.21	0.21	0.21	0.20	0.20	0.21	2.1e-2
58	0.26	0.28	0.22	0.22	0.22	0.22	0.22	0.22	0.22	0.22	0.23	2.3e-2
59	0.42	0.30	0.29	0.28	0.30	0.29	0.28	0.29	0.29	0.28	0.30	4.2e-2
60	0.27	0.23	0.17	0.17	0.17	0.17	0.17	0.17	0.17	0.17	0.19	3.3e-2
61	0.32	0.30	0.28	0.28	0.27	0.27	0.27	0.27	0.27	0.28	0.28	1.6e-2
62	0.27	0.27	0.22	0.22	0.22	0.22	0.22	0.22	0.22	0.22	0.23	2.1e-2
63	0.30	0.26	0.17	0.17	0.17	0.17	0.17	0.17	0.17	0.17	0.19	4.9e-2
64	0.22	0.22	0.16	0.16	0.16	0.16	0.16	0.16	0.16	0.16	0.17	2.7e-2

Table 9: Timing (in seconds) for 10 times of executing (trials) and its mean $\bar{T}$, standard deviation σ_T for 3D with N=128. We varied the number of MPI processes from 1 to 64.

#procs	Trial										$\bar{T}$	σ_T
	1	2	3	4	5	6	7	8	9	10		
1	101.0	101.3	100.9	100.7	101.1	100.8	101.3	101.2	101.4	100.6	101.0	2.7e-1
2	53.19	52.85	53.10	53.40	52.72	53.12	53.25	53.18	53.28	53.27	53.14	2.1e-1
3	38.35	38.45	38.54	38.37	38.41	38.48	38.68	38.58	38.37	38.54	38.48	1.1e-1
4	30.82	30.68	30.66	30.52	30.73	30.98	30.63	30.65	30.93	30.72	30.73	1.4e-1
5	28.35	28.61	28.19	28.28	28.11	28.20	28.44	27.97	28.27	28.43	28.28	1.8e-1
6	25.07	25.29	25.13	24.97	24.79	25.03	25.14	25.28	25.22	25.16	25.11	1.5e-1
7	26.63	26.88	26.85	26.93	26.81	26.98	26.66	26.83	26.94	26.86	26.84	1.1e-1
8	21.96	22.04	22.04	21.92	22.02	22.01	22.03	21.94	21.97	21.93	21.99	4.8e-2
9	21.70	21.79	21.57	21.61	21.68	21.69	21.63	21.61	21.59	21.64	21.65	6.5e-2
10	21.36	21.36	21.42	21.33	21.35	21.35	21.34	21.26	21.31	21.36	21.34	4.1e-2
11	26.09	26.18	26.16	26.12	26.09	26.14	26.18	26.15	26.06	26.20	26.14	4.6e-2
12	19.51	19.56	19.48	19.53	19.48	19.54	19.50	19.55	19.47	19.49	19.51	3.2e-2
13	25.86	26.06	25.97	25.88	26.02	26.07	25.92	25.90	26.04	26.01	25.97	7.8e-2
14	22.17	22.14	22.27	22.20	22.09	22.27	22.12	22.26	22.19	22.26	22.20	6.6e-2
15	19.48	19.52	19.53	19.45	19.50	19.55	19.51	19.51	19.45	19.37	19.49	5.3e-2
16	18.79	18.83	18.78	18.66	18.74	18.72	18.91	18.82	18.91	18.89	18.80	8.3e-2
						.....						
48	6.46	6.47	6.46	6.45	6.48	6.47	6.48	6.45	6.45	6.48	6.47	1.4e-2
49	6.86	6.86	6.86	6.86	6.85	6.85	6.86	6.86	6.87	6.85	6.86	6.7e-3
50	6.34	6.30	6.32	6.32	6.32	6.31	6.33	6.31	6.35	6.33	6.32	1.4e-2
51	7.78	7.78	7.79	7.82	7.84	7.78	7.80	7.93	7.84	7.80	7.82	4.6e-2
52	7.03	7.02	7.06	7.05	7.01	7.12	7.02	7.08	7.03	7.08	7.05	3.6e-2
53	11.74	11.78	11.75	11.75	11.83	11.74	11.72	11.83	11.79	11.72	11.77	4.0e-2
54	5.91	5.90	5.91	5.91	5.92	5.91	5.91	5.90	5.89	5.89	5.91	1.0e-2
55	6.46	6.48	6.46	6.48	6.45	6.43	6.43	6.47	6.41	6.45	6.45	2.3e-2
56	5.86	5.88	5.85	5.86	5.83	5.85	5.82	5.88	5.83	5.87	5.85	2.0e-2
57	7.18	7.15	7.12	7.13	7.14	7.15	7.15	7.11	7.13	7.10	7.14	2.2e-2
58	8.08	8.06	8.01	8.16	8.12	8.07	8.06	8.00	8.06	8.10	8.07	4.7e-2
59	10.88	10.99	11.09	10.97	11.00	11.07	11.03	10.88	10.97	10.97	10.98	6.9e-2
60	5.29	5.28	5.29	5.30	5.30	5.31	5.30	5.29	5.31	5.30	5.30	9.0e-3
61	10.68	10.64	10.64	10.61	10.66	10.67	10.60	10.62	10.65	10.66	10.64	2.8e-2
62	7.56	7.50	7.55	7.53	7.67	7.54	7.54	7.58	7.53	7.54	7.55	4.4e-2
63	5.15	5.14	5.17	5.15	5.17	5.16	5.14	5.19	5.16	5.17	5.16	1.5e-2
64	4.88	4.86	4.87	4.86	4.89	4.87	4.88	4.86	4.85	4.86	4.87	1.2e-2

Table 10: Timing (in seconds) for 10 times of executing (trials) and its mean $\bar{T}$, standard deviation σ_T for 3D with N=256. We varied the number of MPI processes from 1 to 64.

References

1. HTTP://WWW.NEW NPAC.ORG, *Creating a cartesian virtual topology*, 2015.

2. WIKIPEDIA, *Dirichlet boundary condition — wikipedia, the free encyclopedia*, 2014. [Online; accessed 28-September-2015].

3. ——, *Arithmetic mean — wikipedia, the free encyclopedia*, 2015. [Online; accessed 2-October-2015].

4. ——, *Conjugate gradient method — wikipedia, the free encyclopedia*, 2015. [Online; accessed 13-September-2015].

5. ——, *Jacobi method — wikipedia, the free encyclopedia*, 2015. [Online; accessed 13-September-2015].

6. ——, *Speedup — wikipedia, the free encyclopedia*, 2015. [Online; accessed 15-September-2015].

7. ——, *Standard deviation — wikipedia, the free encyclopedia*, 2015. [Online; accessed 2-October-2015].