Generating Gaussian Random Field With Given Autocorrelation Functions using Karhunen-Loève Expansion
APMA 4302 Methods in Computational Science (Fall2020)

Ashutosh Rawat (ar4171), Gurpreet Singh Hora (gh2546) - `@columbia.edu`

12/14/2020

NOTE: For the data and code, please visit the project website: `https://github.com/gh2546/Project_APMA4302`

1 Introduction

In engineering problems, we often deal with the random fields to represent stochastic processes, for instance, turbulence and earthquake modelling. One of the possible ways to generate samples of a stationary random field is the Karhunen–Loève expansion (KLE). KLE can be computed using the covariance kernel modal decomposed on a finite domain. It represents a random field as a linear combination of infinite orthogonal functions. These orthogonal functions represents the eigen-function of the covariance matrix and the coefficients are the eigen-values of the covariance matrix. Typically, the size of domain is much larger than the length scale called correlation length and due to which random field generation process becomes computationally expensive. To overcome this, we can parallelize the KLE on distributed memory system. In parallel version of the KLE, each processor solves the eigen-values and the eigen-functions of the covariance matrix on a small sub-domain and it helps in reducing the computational cost. Further, to obtain continuous sample of the random fields generated using the prescribed auto-correlation function, we can gather eigen-values and eigen-functions from different processors and can combine them in a linear fashion.

For this project, we utilized an open-source library named deal.II, which is implemented over C++.

2 Governing Equations

Any gaussian random field $\{\Xi(\mathbf{x}), \mathbf{x} \in \mathbb{R}^n\}$ generated using KLE on a computational domain can be represented as follows:

$$\Xi_i(\mathbf{x}) = \sum_{k=1}^{q_i} \sqrt{\lambda_k}\eta_k\psi_k(\mathbf{x}) \tag{1}$$

where, η_k is a set of independent Gaussian random variables, λ_k and $\psi_k(\mathbf{x})$ are the non-negative eigen-values and the associated orthonormal eigen-functions for the given autocorrelation function. In addition to this, eigenfunctions must also satisfy the Fredholm integral equation of the second kind.

$$\int_\Omega \mathbf{R}(\mathbf{x}, \mathbf{x}')\psi_k(\mathbf{x}')\mathrm{d}\mathbf{x}' = \lambda_k\psi_k(\mathbf{x}) \tag{2}$$

where, $\mathbf{R}$ is the covariance kernel or autocorrelation function. For any arbitrary autocorrelation function, this integral equation can be solved by using a standard Galerkin formulation given a finite set of basis functions $\{\phi_j(\mathbf{x})\}$. Each eigen-function can be approximated as:

$$\psi_k(\mathbf{x}) \approx \sum_{j=1}^{N_{\text{node}}} \alpha_j^{(k)}\phi_j(\mathbf{x}) \tag{3}$$

By substituting (3) in (2) we will get:

$$\sum_{j=1}^{N_{\text{node}}} \alpha_j^{(k)} \int_\Omega\int_\Omega \mathbf{R}(\mathbf{x}, \mathbf{x}')\phi_j(\mathbf{x}')\phi_l(\mathbf{x})\mathrm{d}\mathbf{x}'\mathrm{d}\mathbf{x} = \sum_{j=1}^{N_{\text{node}}} \alpha_j^{(k)} \int_\Omega \lambda_k\phi_j(\mathbf{x})\phi_l(\mathbf{x})\mathrm{d}\mathbf{x} \tag{4}$$

Considering the above equation for eigen-functions leads to the generalized eigenvalue problem $[K]\,[A] = [\Lambda]\,[M]\,[A]$, where, $[A]_{jl} = \alpha_j^{(l)}$, $[\Lambda]_{jl} = \lambda_j\delta_{jl}$ (where, δ_{jl} is the Kronecker delta), and $[K]$ and $[M]$ are the matrices with the entries:

$$[K]_{jl} = \int_\Omega\int_\Omega \mathbf{R}(\mathbf{x}, \mathbf{x}')\phi_j(\mathbf{x}')\phi_l(\mathbf{x})\mathrm{d}\mathbf{x}'\mathrm{d}\mathbf{x}$$
$$[M]_{jl} = \int_\Omega \lambda_k\phi_j(\mathbf{x})\phi_l(\mathbf{x})\mathrm{d}\mathbf{x}$$

3 Numerical Implementation

For this report, a gaussian field $\{\Xi(\mathbf{x}), \mathbf{x} \in \mathbb{R}^n\}$ has been generated for 2D domain $\Omega = [-1, 1] \times [-1, 1]$, using following autocorrelation function:

$$\mathrm{R}(\mathbf{x}, \mathbf{x}') = \exp\left(\frac{-||\mathbf{x} - \mathbf{x}'||}{2l^2}\right) \tag{5}$$

where, l is the correlation length, it's value kept as 0.05. For the finite element formulation bi-linear shape functions $\{\phi(\mathbf{x})\}$ are used to approximate the eigen-functions $\{\psi(\mathbf{x})\}$. The

$[K]$ and $[M]$ matrices are computed at each quadrature point and assembled by using **FE_Q** class in deal.II.

After assembling the $[K]$ and $[M]$ matrices the generalized eigenvalue problem is solved using the **SLEPc Krylov-Schur solver** interface in deal.II. After obtaining the eigen-values λ_k and eigen-functions $\psi_k(\mathbf{x})$, a set of k independent Gaussian random variable $\{\eta_k\}$ are generated and using (1) a random field $\{\Xi(\mathbf{x})\}$ can be easily obtained.

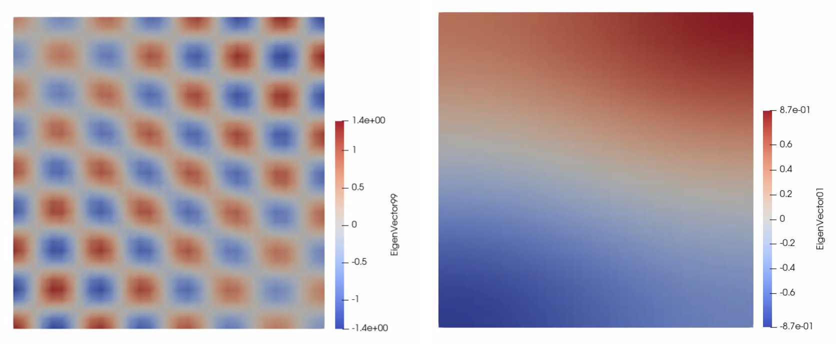

Figure: Plots of Eigen-functions (corresponding to smallest eigen-value (on left) and largest eigen-value (on right))

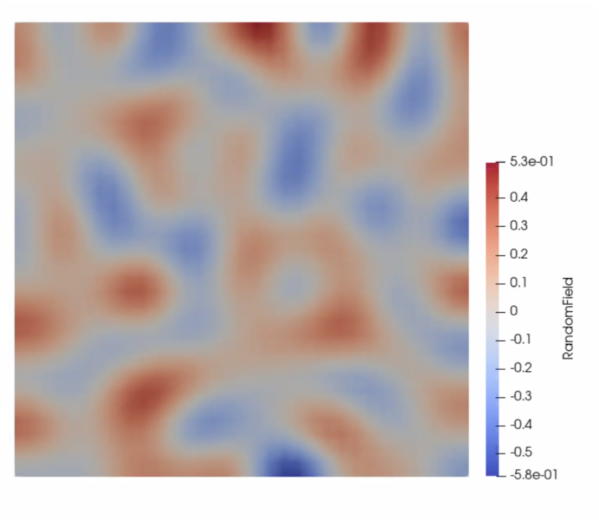

Figure: Plot of Gaussian random field generated using 100 eigen-functions

4 Brief Review of Code

The code is written using **deal.II** library in C++. It leverages in-built classes and data-structures of deal.II for both FE implementation and parallelization. deal.II automatically takes care of all the parallel communication in a very efficient way.

We first set up a class template for our problem. It will be main class of the code and creates different class objects for generating the grid, assembling and solving the system, creating a random field and outputting the results. The main class also calls MPI class object for parallel communication.

```
deal.II Mesh <>
class ParallelKL \\name of the class
{
public:
  ParallelKL();
  void run();
private:
  void make_grid();
  void setup_system();
  void assemble_system();
  void solve();
  void create_random_field();
  void stats_for_random_field();
  void output_results() const;
  void output_results_parallel();

  MPI_Comm mpi_communicator;

  parallel::distributed::Triangulation<2> triangulation;

  FE_Q<2>          fe;
  DoFHandler<2>    dof_handler;

  IndexSet         locally_owned_dofs;
  IndexSet         locally_relevant_dofs;

  AffineConstraints<double> constraints;

  PETScWrappers::MPI::SparseMatrix system_mass_matrix;
  PETScWrappers::MPI::SparseMatrix system_stiffness_matrix;

  PETScWrappers::MPI::Vector randomfield_vector;

  std::vector<double> eigenvalues;
  std::vector<PETScWrappers::MPI::Vector> eigenvectors;

  std::vector<double> normalized_gaussian;
  std::default_random_engine generator;
  std::normal_distribution<double> distribution;

  unsigned int this_mpi_process, n_mpi_processes;

  ConditionalOStream pcout;
  TimerOutput        computing_timer;
};
```

The **parallel::distributed::Triangulation** class creates the mesh in parallel, **DoFHandler** class handles the DoFs and contains information about the global indices of DoFs in *locally owned cells* and *ghost cells* (described in later section). The **IndexSet** class stores the subset of indices from the larger set in DoFHandler. The class variable locally_owned_dofs store the indices of DoFs which are in *locally owned cells* and locally_relevant_dofs stores indices of DoFs which are in locally owned subdomain as well as the adjacent *ghost cells*. This code uses **PETSc** for linear algebra. The **PETScWrapper** class provides the ability to do this and can be easily used in parallel with **PETScWrapper::MPI**.

The following part shows how the matrices are $[K]$ here called *system_ stiffness_ matrix* and $[M]$ here called *system_ mass_ matrix* are being

```
void ParallelKL::assemble_system()
{
  TimerOutput::Scope t(computing_timer, "assembly");

  system_stiffness_matrix = 0.;
  system_mass_matrix = 0.;

  QGauss<2> quadrature_formula(fe.degree + 1);

  FEValues<2> fe_values(fe, quadrature_formula,
              update_values | update_gradients |
              update_JxW_values | update_quadrature_points);

  const unsigned int dofs_per_cell = fe.dofs_per_cell;

  FullMatrix<double> cell_mass_matrix(dofs_per_cell,
                                      dofs_per_cell);
  FullMatrix<double> cell_stiffness_matrix(dofs_per_cell,
                                           dofs_per_cell);

  std::vector<types::global_dof_index>
                        local_dof_indices(dofs_per_cell);

  for (const auto &cell : dof_handler.active_cell_iterators())
  { if (cell->is_locally_owned())
    {
      fe_values.reinit(cell);
      cell_mass_matrix = 0.;
      cell_stiffness_matrix = 0.;

     for(const unsigned int q_index:
        fe_values.quadrature_point_indices())
     {
       Point<2> quad_pointq=fe_values.quadrature_point(q_index);
       for (const unsigned int i:fe_values.dof_indices())
        for (const unsigned int j : fe_values.dof_indices())
          cell_mass_matrix(i, j) +=
            (fe_values.shape_value(i, q_index)*//grad phi_i(x_q)
            fe_values.shape_value(j, q_index)*//grad phi_j(x_q)
            fe_values.JxW(q_index));          //dx

              for (const unsigned int l_index :
                    fe_values.quadrature_point_indices())
                {
                  Point<2> quad_pointl =
                    fe_values.quadrature_point(l_index);
                  const double point_distance =
                    quad_pointl.distance(quad_pointq);

                  for (const unsigned int i:
                        fe_values.dof_indices())
                    for (const unsigned int j:
                      fe_values.dof_indices())
                       cell_stiffness_matrix(i, j) +=
                      (exp(-0.5*point_distance/(0.05*0.05))*
                     fe_values.shape_value(i, q_index)*
                     fe_values.shape_value(j, l_index)*
                fe_values.JxW(q_index)*fe_values.JxW(l_index));
                }
     }
     cell->get_dof_indices(local_dof_indices);

    constraints.distribute_local_to_global(cell_stiffness_matrix,
                                           local_dof_indices,
                                           system_stiffness_matrix);

    constraints.distribute_local_to_global(cell_mass_matrix,
                                           local_dof_indices,
                                           system_mass_matrix);
  }
 }

system_mass_matrix.compress(VectorOperation::add);
system_stiffness_matrix.compress(VectorOperation::add);
}
```

The sparse matrices *system_ stiffness_ matrix* and *system_ mass_ matrix* are assembled in parallel, these both matrices have information about all the *locally_ owned_ dofs* and *locally_ relevant_ dofs*, and takes care of all the communication as they object of class

PETScWrappers::MPI. We iterate over all the *active cells* and only for the cells which are locally owned we iterate over all the quadrature points in it and find the elements of the $[K]$ and $[M]$ matrices. These values are stored in local cell matrix and then finally distributed to the global matrices. Finally we compresses the sparsity pattern as it allows the resulting matrix to be used in all other operations where before only assembly functions were allowed.

The next section shows how the generalized eigenvalue problem is solved using **SLEPScWrappers** in deal.II, this carries the same behaviour as **PETScWrappers** and take care of all parallel communication. We have used **Krylov-Schur** technique to solve the generalized eigenvalue problem.

```
void ParallelKL::solve()
{
  const unsigned int num_eigenpairs_requested = 100;

  eigenvalues.resize(num_eigenpairs_requested);
  eigenvectors.resize(num_eigenpairs_requested);

  normalized_gaussian.resize(num_eigenpairs_requested);

  for (unsigned int i = 0; i < num_eigenpairs_requested; ++i)
    eigenvectors[i].reinit(locally_owned_dofs,
                  locally_relevant_dofs, mpi_communicator);

  SolverControl eigen_solver_control (10000, 1e-10);

  SLEPcWrappers::SolverKrylovSchur
     eigensolver(eigen_solver_control, mpi_communicator);

  eigensolver.set_which_eigenpairs(EPS_LARGEST_REAL);

  eigensolver.set_problem_type(EPS_GHEP);

  pcout << "Beginning Eigensolve..." << std::endl;
  eigensolver.solve(system_stiffness_matrix,
                    system_mass_matrix,
                    eigenvalues,
                    eigenvectors,
                    num_eigenpairs_requested);

  for (unsigned int i = 0; i < num_eigenpairs_requested; i++)
  {
    double temporary_sample = 0.0;

    if (this_mpi_process == 0)
      temporary_sample = distribution(generator);
    double temporary_sum = 0.0;

    MPI_Allreduce (&temporary_sample,
            &temporary_sum, 1, MPI_DOUBLE,
            MPI_SUM, mpi_communicator);

    normalized_gaussian[i] = temporary_sum;

  }
}
```

5 deal.II

deal.II is an open-source C++ library, used for solving partial differential solution using finite element method. It provides support to both PETSc and Trilinos framework for solving sparse linear algebra problems. It is also highly efficient in solving problems across thousands of processors. The important aspects of FEM software/library is the way it handles the mesh, degree of freedom, global and local matrices, sparsity pattern of the matrices and solution vector. deal.II has inbuilt data structures and class templates to handle above things so that it becomes very easy and efficient for the user to implement finite element procedures.

In the distributed parallel implementation in deal.II each machine has the information of entire mesh and degree of freedoms (DoFs), but shared information of global matrix, sparsity pattern and solution vector. In a particular machine each processor stores only a share of the cells and DoFs. The processors do not have any knowledge of entire mesh, matrix or solution.

5.1 Distributed Mesh

deal.II uses **p4est** algorithm for generating 2D and 3D mesh topology. Many finite element libraries are restricted in scalability due to the requirement of storing global mesh structure on each processor. deal.II overcame this bottleneck by using distributed mesh storage. Each processor still stores the entire global mesh, but only the part of mesh which is "locally owned" by the processor is stored in fine resolution, the rest of the mesh is stored in coarser form, which drastically reduces the memory footprint while scaling the problem.

Though deal.II uses distributed mesh based upon p4est, but it completely modifies theses meshes to further increase the efficiency. deal.II employs rich data structures for the mesh, as FE algorithms requires information about actual geometric location of vertices, boundary indicators and material properties etc. It creates its own local mesh using the information from the locally stored part of mesh created by p4est. The mesh created by deal.II has different types of cells:

- *Active cells*: These cover entire domain. The active cells which are in the local mesh owned by current processor are called *locally owned active cells*.

- *Ghost cells*: Active cells which are adjacent to the *locally owned active cell.* These are owned by different a processor.

5.2 Distributed DOF Handling

In deal.II to connect the DOFs to the meshes **DoFHandler** class is used. It allocates global number for each of the DoFs located on different vertices, lines, faces and cell. In parallel, the task of assigning a global index to each DoF defined on the mesh which is locally owned is very tricky. deal.II uses a unique algorithm to enumerate DoFs in parallel. The algorithm works in following way:

1. Initialize the indices of DoFs on all the active cell with invalid value (eg. -1).
2. Loop over all the cells belonging to set of cells locally owned by processor p and flag the indices of all the DoFs to a valid value (eg. 0).
3. If a cell belongs to the set of ghost cells in processor p, and its owner q is less than p, then reset the indices of the DoF on this cell to invalid value.
4. Loop over all the locally owned cells and assign the indices in ascending order to all the DoFs marked as valid. Let n_p be the number of indices assigned.
5. Using **MPI_Allgather** communicate this n_p to all the processors. Shift all the indices of DoF by $\sum_{i=0}^{p-1} n_i$. All the indices have been enumerated, but processor p still may not know the correct indices of degrees of freedom on the interface between its cells and those owned by other processors, as well as the indices on ghost cells.

To know the indices of DoFs on *ghost cells* and interface of *local* and *ghost cells*, the indices of DoFs on *locally owned cells* are communicated to other processors by following the algorithm:

- Flag all the vertices of cells on the set of locally owned cells.
- Loop over all the cells in the set of ghost cells in processor p, and for the vertices in a cell flagged in above step, store it.
- Loop over all cells in the set of locally owned cells, and if it shares one of its vertices with the ghost cell q, then add the information of *cell_ id* and *indices of DoFs* to a list which will be sent to q.
- After making such a list, send it to processor q.
- Processor p will also receive such list from all the processors, and update the indices of all DoFs of the cells in the list received.
- Repeat the above steps so that the indices of DoFs which are not known previously due to the parallel operations are also updated.

Now each processor knows the global index of all the DoFs that are in its *local* and *ghost* cells, and also on the interface between them.

5.3 Setting Up and Solving Linear Systems

deal.II uses **PETSc** and **Trilinos** wrappers for parallel linear algebra. As both of these libraries handles sparse matrices very efficiently. deal.II takes advantage of these libraries and creates objects with special data structure that efficiently copy the information of row and column indices into **PETSc** and **Trilinos** matrix classes. After building up the sparsity pattern the assembling of matrix takes place in the usual manner by taking into contribution of *locally owned* cells in each processor and communicating between all the processors to transfer the local entries into global matrix. **PETSc** and **Trilinos** automatically takes care of all these communications.
The resulting linear system is then solved by these libraries using the variety of solvers and commonly used preconditioners.

6 Computational Domain

To calculate the gaussian random field, a suite of simulations has been carried out in the computational domain of size $[-L_x, L_x] \times [-L_y, L_y]$ with $L_x = 1$m and $L_y = 1$m. The computational domain has been discretized by 66049 and 1050625 degrees of freedom and referred as coarse and fine mesh cases respectively. 100 eigen-values are linearly combined to generate the gaussian random field. All performance analysis reported in this report are performed as per this computational domain.

7 Performance Analysis

7.1 Profiling

Profiling is a technique employed by developers to analyze the complexity of different components of the algorithm. The complexity can be measured either in terms of the time an algorithm takes to execute or in terms of the memory usage. The analysis is very useful to identify the performance bottleneck sections of the algorithm and later on developers can work on optimizing the bottleneck sections of the algorithm to get a more optimized and efficient algorithm.

In the project, we have developed an algorithm to generate 2-D random gaussian fields using KL expansions. It involves different processes such as setup of the required arrays, assembly of the arrays from different processes, generating 2-D grid for the aforementioned computational domain, solving the set of equations to get eigenvalues of large matrices and then outputting the results. To perform this, we extracted the individual wall clock time of the processes and plotted them in the figure below. From the plot, it is evident that as we increase the number of processors, time required to perform different tasks reduces. In addition to this, we can observe that time taken by most of the components are negligible as compared to the solver and outputting the results. It seems that most of the algorithm components are already optimized and no need to work on them. The outputting process can be improved if we switch to binary format instead of ASCII. For solving the set of equations, we are using the KrylovSchur algorithm, a robust and an efficient algorithm to find few eigenvalues of large matrices from the deal.II library. The algorithm is already efficiently implemented and there is no room for further improvement.

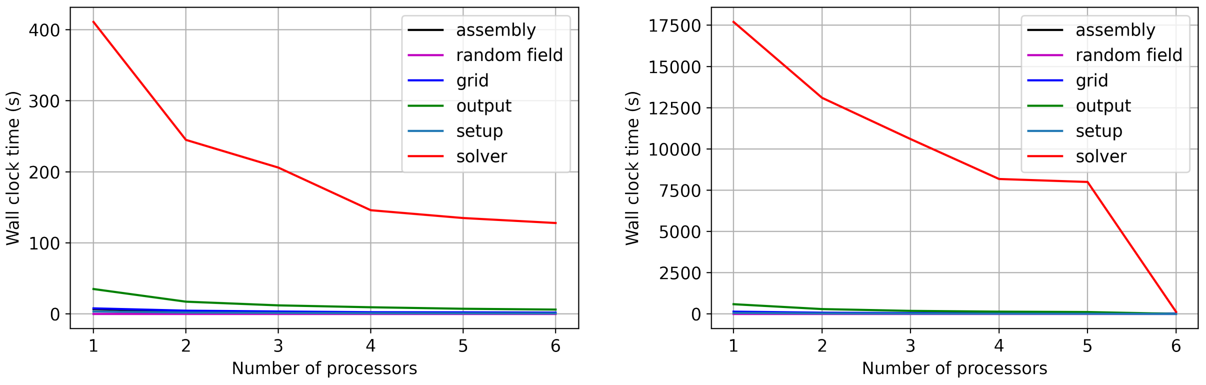

Figure: Plot of profiling for coarse mesh (on left) and fine mesh (on right) cases

7.2 Scalability

Scalability is a parallel performance testing function that measures the ability of an algorithm to deliver with the more computational resources (cores/ processors/ threads). The two most commonly used scalability functions are strong scaling (aka Amdhal's law) and weak scaling (aka Gustafon's law). In the strong scaling, we fix the size of the problem and check the algorithm performance with an increasing number of computational resources. It gives us an estimate of how parallelism can reduce the execution time of the algorithm for the given problem size. It may also be useful in finding an optimized number of processors for large-scale numerical experiments. On the other hand, in the weak scaling analysis, we vary the problem size as well as the number of computing resources such that the load on each processor or thread or core will remain constant. This provides an estimate of how much long it takes to complete the task with and without parallelism. This is very useful for the large memory-bound applications i.e. one core or processor or thread won't be able to satisfy the memory requirement.

To perform the strong scaling analysis, we carried numerical experiments on the aforementioned computational domain and varying programmatically the number of processors. Typically, the number of processors increased by a factor of 2 i.e. 1, 2 , 4 and so on, but due to computational resource restrictions on our system, we varied it from 1 to 6. From the strong scaling and strong speed up plot, it can be clearly observed that the clock wall time has been first reduced significantly with the number of processors and then the curve becomes flat. The flattening of the curve indicates that further computational resources won't improve the performance of the algorithm. This trend also aligns with the ideal trend i.e. for a fixed size problem, the time taken by the algorithm to complete the task will reduce upto certain number of processors and then becomes constant. It should be noted that after certain computational resources the performance of the algorithm is not improving because the ratio of the parallel fraction of the code and the number of processors becomes a small number and further parallelism won't help. Sometimes, we may also observe the worsening of the performance after a flat curve and it mainly due to the high cost of communication among processors.

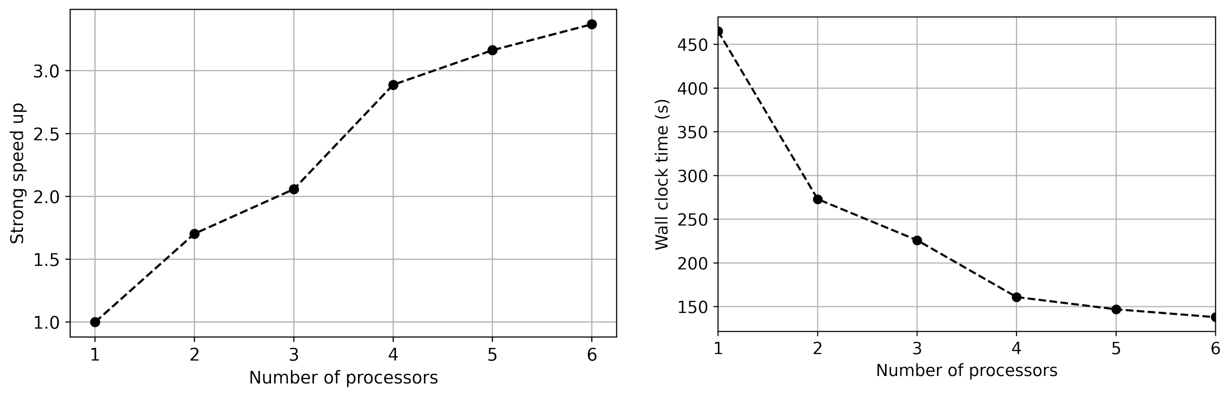

Figure: Plot of Strong speed up (on left) and Strong scaling (on right) for coarse mesh

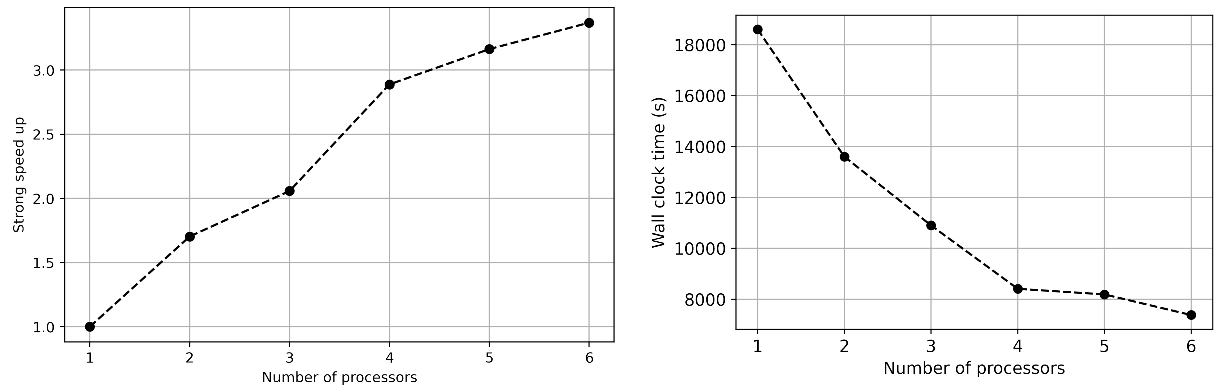

Figure: Plot of Strong speed up (on left) and Strong scaling (on right) for fine mesh

We also performed the weak scaling analysis of the algorithm. We carried numerical experiments for the above mentioned computational domain and this time we vary the number of eigenvalues as well as the number of processors, such that each processor have same load of work. From the weak scaling or weak efficiency plot, it can be seen that to perform the same task per processor in parallel takes more time as compared to do the same without parallelism. This may be attributed to the high communication cost among processors. Another reason may be the less memory bandwidth available in the parallelism as compared to without parallelism scenario.

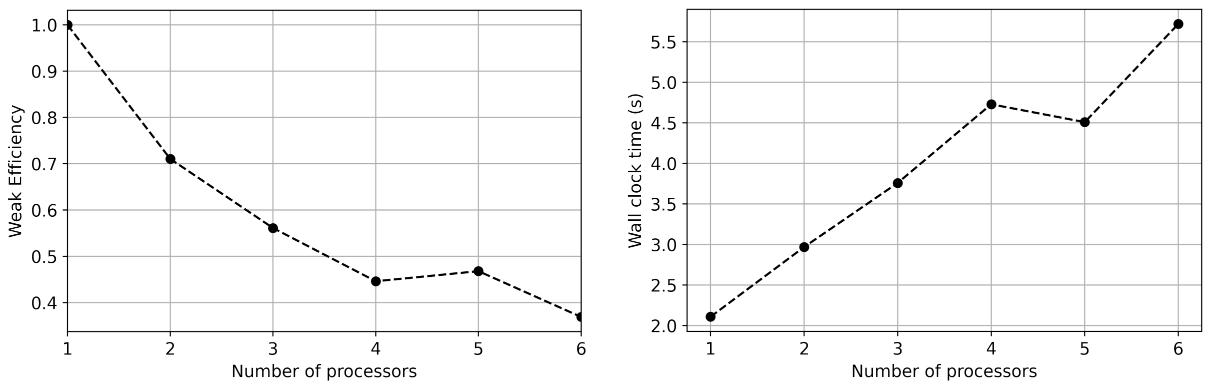

Figure: Plot of Weak efficiency (on left) and Weak scaling (on right) for constant workload on a processor.

7.3 Roofline model

Roofline model is a way to represent the performance estimates of the compute kernel. It can also be used to represent the performance of algorithms on the given architecture. NOTE: Architecture information has been provided in the appendix. Below, we have plotted the performance of the floating point operations as a function of machine peak performance. The point where the bound based on bandwidth and bound based on peak performance meets is known as ridge point and it's approximate value is 20 FLOP/Byte for our case.

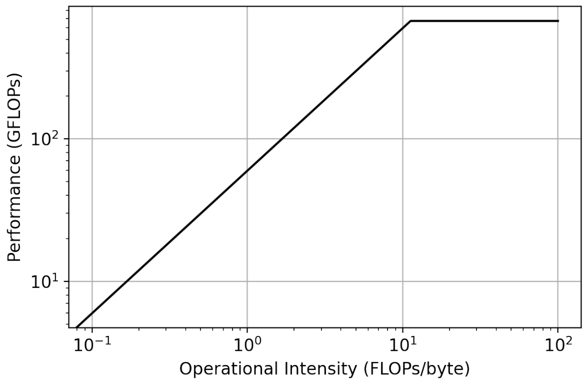

Figure: Plot of Roofline model for Xeon(R)

8 References

1. Bangerth, W., Burstedde, C., Heister, T., Kronbichler, M. (2012). Algorithms and data structures for massively parallel generic adaptive finite element codes. ACM Transactions on Mathematical Software (TOMS), 38(2), 1-28.
2. Bangerth, W., Hartmann, R., Kanschat, G. (2007). deal. II—a general-purpose object-oriented finite element library. ACM Transactions on Mathematical Software (TOMS), 33(4), 24-es.
3. van Rossum, G. (1995). Python tutorial, Technical Report CS-R9526, Centrum voor Wiskunde en Informatica (CWI), Amsterdam.".
4. Hu, T., Guilleminot, J., Dolbow, J. E. (2020). A phase-field model of fracture with frictionless contact and random fracture properties: Application to thin-film fracture and soil desiccation. Computer Methods in Applied Mechanics and Engineering, 368, 113106.
5. Panunzio, A. M., Cottereau, R., Puel, G. (2018). Large scale random fields generation using localized Karhunen–Loève expansion. Advanced Modeling and Simulation in Engineering Sciences, 5(1), 20.
6. Hernandez, V., Roman, J. E., Tomas, A. (2007, December). A parallel Krylov-Schur implementation for large Hermitian and non-Hermitian eigenproblems. In PAMM: Proceedings in Applied Mathematics and Mechanics (Vol. 7, No. 1, pp. 2020083-2020084). Berlin: WILEY-VCH Verlag.
8. Ghanem, R. G., Spanos, P. D. (2003). Stochastic finite elements: a spectral approach. Courier Corporation. 9. Le Maître, O., Knio, O. M. (2010). Spectral methods for uncertainty quantification: with applications to computational fluid dynamics. Springer Sci-

ence Business Media. 10.“Lecture2: Performance Analysis” David Bindel `https://www.cs.cornell.edu/courses/cs5220/2020fa/schedule.html` Accessed 14 Dec. 2020.
11.“High Performance Computing for Science and Engineering” Petros Koumoutsakos `https://www.cse-lab.ethz.ch/teaching/hpcse-i_hs18/` Accessed 14 Dec. 2020
12.“hw1_theory.ipynb Methods in Computational Science” Kyle T. Mandli

9 Acknowledgment

We would like to thanks Dr. Mandli for his help, patience, energy, suggestions, and time he invested in lectures and guiding us for the project during the semester. We both truly appreciate your help.

10 Appendix

10.1 Architecture details

All the simulations are carried out on Intel(R) Xeon(R) CPU E5-1650 V2 processors. We have in total 6 cores on the system. The clock rate for the processor is 2.50 GHz and instruction set is 64 bit. The type of the instruction set is AVX-512 (16 single precision floating number). The system has DDR4-1866 RAM with 4 slots. Following are the calculations for π and β.

$$\begin{aligned} \pi &= \text{clock rate} \times \text{cores} \times < ashutosh > \\ &= 3.50 \times 6 \times 16 \times 2 \\ &= 672 \quad \text{GFLOP/s} \\ \\ \beta &= 1.866 \times 8 \times 4 \\ &= 59.712 \end{aligned}$$

10.2 Automated Python script to launch the series of the simulations

```
import os
import sys
import shutil
import subprocess
import time
import re
from datetime import date
from datetime import datetime

num_procs = [6, 5, 4, 3, 2, 1]
path_dir = "../KLexpansion_results/"
```

```
code_dir = "../KLexpansion"
path_dir_list = []
exect_command = []

for nprocs in num_procs:
    path_dir_list.append(path_dir + '{0}'.format(nprocs))
    exect_command.append("mpirun -n {0} klexpansion > out.log".format(
                                                 nprocs))


def purge(dir, pattern):
    for f in os.listdir(dir):
        if re.search(pattern, f):
            os.remove(os.path.join(dir, f))

for i in range(len(exect_command)):
    os.chdir(code_dir)
    print("*************************************")
    print("Launching new job. \n")
    print("Code is stored in the dir:{}".format(code_dir))
    print("Command executed on the shell is: {0} ".format(exect_command[i]
                                                 ))
    print("Job has been launched on {0}.".format(datetime.now().strftime("
                                                 %d/%m/%Y %H:%M:%S")))
    p = subprocess.Popen([exect_command[i]], shell = True)
    p.wait()
    print("Job has been completed on {0} and output has been stored in out
                                                 .log file.".format(datetime.now()
                                                 .strftime("%d/%m/%Y %H:%M:%S")))
    print("Copying the data from code directory:{0} to result directory:{1
                                                 }".format(code_dir, path_dir_list
                                                 [i]))
    shutil.copytree(code_dir, path_dir_list[i])
    print("Cleaning the code directory and preparing it for new job.")
    purge(code_dir, "Solution")

    print("Putting the scheduler on sleep for 10 seconds.")
    print("*************************************")
    time.sleep(10)
```

10.3 Plotting python script

```
import os
import matplotlib.pyplot as plt
import numpy as np
import matplotlib as mpl
os.chdir("../figures/coarse_resolution/")
num_procs = [1,2,3,4,5,6]
total_time_strong = [465, 273, 226, 161, 147, 138]
assembly_1 = [6.91, 3.63, 2.58, 1.87, 1.52, 1.29]
gen_rand_field = [0.00961, 0.00502, 0.00345, 0.00277, 0.00224, 0.00202]
```

```
grid_gen = [7.9, 4.65, 3.49, 2.48, 2.33, 1.94]
out = [35.1, 17.3, 12.1, 9.43, 7.32, 6.17]
setup = [3.5, 1.98, 1.49, 1.05, 0.913, 0.824]
solver = [411, 245, 206, 146, 135, 128]
stats = [0.682, 0.353, 0.241, 0.182, 0.146,0.127]
plt.figure(figsize=(6,4), dpi=400)
speed_up = [465/465, 465/273, 465/226, 465/161, 465/147, 465/138]
plt.plot(num_procs,speed_up, "ko--")
plt.xlabel(r"Number of processors")
plt.tight_layout()
plt.ylabel("Strong speed up")
plt.grid()
mpl.rcParams.update({'font.size': 12})
plt.savefig("speed_up_strong", dpi=400)
plt.show()
plt.figure(figsize=(6,4), dpi=400)
plt.plot(num_procs, total_time_strong, "ko--")
plt.xlabel(r"Number of processors")
plt.ylabel(r"Wall clock time (s)")
plt.tight_layout()
mpl.rcParams.update({'font.size': 12})
plt.grid()
plt.savefig("strong_scaling", dpi=400)
plt.show()
total_time_weak = [2.11/2.11, 2.11/2.97, 2.11/3.76, 2.11/4.73, 2.11/4.51,
                                     2.11/5.72]
num_procs = [1,2,3,4,5,6]
plt.figure(figsize=(6,4), dpi=400)
plt.plot(num_procs, total_time_weak, "ko--")
plt.xlabel(r"Number of processors")
plt.ylabel(r"Weak Efficiency")
plt.tight_layout()
mpl.rcParams.update({'font.size': 12})
plt.grid()
plt.savefig("weak_scaling_speedup", dpi=400)
plt.show()
total_time_weak = [2.11, 2.97, 3.76, 4.73, 4.51, 5.72]
num_procs = [1,2,3,4,5,6]
plt.figure(figsize=(6,4), dpi=400)
plt.plot(num_procs, total_time_weak, "ko--")
plt.xlabel(r"Number of processors")
plt.ylabel(r"Wall clock time (s)")
plt.tight_layout()
mpl.rcParams.update({'font.size': 12})
plt.grid()
plt.savefig("weak_scaling", dpi=400)
plt.show()
plt.figure(figsize=(6,4), dpi=400)
plt.plot(num_procs, assembly_1, color='k',label="assembly")
plt.plot(num_procs, gen_rand_field, color='m', label="random field")
plt.plot(num_procs, grid_gen, color='b', label="grid")
plt.plot(num_procs, out, color='g',label="output")
plt.plot(num_procs, setup, label="setup")
plt.plot(num_procs, solver,color='r', label = "solver")
plt.xlabel(r"Number of processors")
plt.ylabel(r"Wall clock time (s)")
plt.tight_layout()
mpl.rcParams.update({'font.size': 12})
plt.grid()
plt.legend(loc='best')
plt.savefig("profiling", dpi=400)
plt.show()
pi = 672
beta = 59.712
I = np.linspace(0, 100, 1000)
P = []
for i in I:
    p = beta*i
    if p >= pi:
        P.append(pi)
    else:
        P.append(p)

plt.figure(figsize=(6,4), dpi=400)
plt.loglog(I, P, "k")
plt.xlabel(r"Operational Intensity (FLOPs/byte)")
plt.ylabel(r"Performance (GFLOPs)")
plt.grid()
plt.tight_layout()
plt.savefig("roof_line", dpi=200)
plt.show()
os.chdir("../figures/fine_resolution/")
num_procs = [1,2,3,4,5,6]
total_time_strong = [1.86e+04, 1.36e+04, 1.09e+04, 8.41e+03, 8.19e+03, 7.
                                        38e+03]
assembly_1 = [123, 61.7, 39.5, 29.8, 24, 20.1]
gen_rand_field = [0.22, 0.14, 0.119, 0.117, 0.117, 0.117]
grid_gen = [140, 78.3, 50.9, 37.6, 33.2, 26.5]
out = [592, 296, 187, 141, 116, 6.17]
setup = [62.1, 34.2, 22.4, 16.6, 13.9, 0.824]
solver = [1.77e+04, 1.31e+04, 1.06e+04, 8.18e+03, 8e+03, 128]
stats = [12.2, 6.09, 3.78, 2.97, 2.4, 0.127]
plt.figure(figsize=(6,4), dpi=400)
speed_up = [465/465, 465/273, 465/226, 465/161, 465/147, 465/138]
plt.plot(num_procs,speed_up, "ko--")
plt.xlabel(r"Number of processors")
plt.tight_layout()
plt.ylabel("Strong speed up")
plt.grid()
mpl.rcParams.update({'font.size': 12})
plt.savefig("speed_up_strong", dpi=400)
plt.show()
plt.figure(figsize=(6,4), dpi=400)
plt.plot(num_procs, total_time_strong, "ko--")
plt.xlabel(r"Number of processors")
plt.ylabel(r"Wall clock time (s)")
plt.tight_layout()
mpl.rcParams.update({'font.size': 12})
```

```
plt.grid()
plt.savefig("strong_scaling", dpi=400)
plt.show()
total_time_weak = [2.11/2.11, 2.11/2.97, 2.11/3.76, 2.11/4.73, 2.11/4.51,
                                      2.11/5.72]
num_procs = [1,2,3,4,5,6]
plt.figure(figsize=(6,4), dpi=400)
plt.plot(num_procs, total_time_weak, "ko--")
plt.xlabel(r"Number of processors")
plt.ylabel(r"Weak Efficiency")
plt.tight_layout()
mpl.rcParams.update({'font.size': 12})
plt.grid()
plt.savefig("weak_scaling_speedup", dpi=400)
plt.show()
total_time_weak = [2.11, 2.97, 3.76, 4.73, 4.51, 5.72]
num_procs = [1,2,3,4,5,6]
plt.figure(figsize=(6,4), dpi=400)
plt.plot(num_procs, total_time_weak, "ko--")
plt.xlabel(r"Number of processors")
plt.ylabel(r"Wall clock time (s)")
plt.tight_layout()
mpl.rcParams.update({'font.size': 12})
plt.grid()
plt.savefig("weak_scaling", dpi=400)
plt.show()
plt.figure(figsize=(6,4), dpi=400)
plt.plot(num_procs, assembly_1, color='k',label="assembly")
plt.plot(num_procs, gen_rand_field, color='m', label="random field")
plt.plot(num_procs, grid_gen, color='b', label="grid")
plt.plot(num_procs, out, color='g',label="output")
plt.plot(num_procs, setup, label="setup")
plt.plot(num_procs, solver,color='r', label = "solver")
plt.xlabel(r"Number of processors")
plt.ylabel(r"Wall clock time (s)")
plt.tight_layout()
mpl.rcParams.update({'font.size': 12})
plt.grid()
plt.legend(loc='best')
plt.savefig("profiling", dpi=400)
plt.show()
```

10.4 deal.ii script to carry out gaussian field simulations

Submitted on Github and courseworks.