Parallelizing the Gauss-Seidel Algorithm by MPI

IN3200/IN4200 Obligatory assignment 2, Spring 2025

Note: This is one of the two obligatory assignments that both need to be approved so that a student is allowed to take the final exam. (The grade of the final exam is otherwise *not* related to these two assignments.)

Note: Discussions between the students are allowed, but each student should write her/his own implementations. The details about the submission can be found in Section 3.3.

1 Objectives

This obligatory assignment has the following objectives:

- The student is given an opportunity to consolidate the understanding about "wavefront" parallelization (described in Section 6.3 of the textbook).
- The student is given a hands-on exercise on MPI programming.
- The student is given an opportunity to run parallel programs on a parallel computer.

2 Introduction

The *Gauss-Seidel* algorithm, in one of its 3D formulations, is introduced in Section 6.3 of the textbook. In this assignment, we assume a 3D array of size `kmax`×`jmax`×`imax`, where `kmax` denotes the total number of points in the `k`-direction (including the two boundary points), and similarly for `jmax` and `imax`. The computation over the 3D array is done by a number of iterations, where the computational work per iteration can be implemented in the following "normal" manner:

```
void GS_iteration_normal (int kmax, int jmax, int imax, double ***phi)
{
  int k,j,i;
  for (k=1; k<kmax-1; k++)
    for (j=1; j<jmax-1; j++)
      for (i=1; i<imax-1; i++)
        phi[k][j][i] = (phi[k-1][j][i] + phi[k][j-1][i]
                       +phi[k][j][i-1] + phi[k][j][i+1]
                       +phi[k][j+1][i] + phi[k+1][j][i])/6.0;
}
```

Note: The computation happens only on the interior points, so the boundary points remain unchanged. In each direction, there is dependency betweeen the loop indices, thus parallelization can *not* be directly applied. However, as explained in Section 6.3 of the textbook, there are other ways of traversing the 3D array that respect the dependency while producing the *same* computational result. One idea is to traverse the 3D array as successive *wavefronts*, where the computational work on each wavefront can be simultaneously carried out by multiple processes (in MPI) or threads (in OpenMP).

One coarse-grained parallelization strategy is to divide the computational points in the `j`-direction, that is, `j=1,2,...,jmax-2`, into chunks. While the first chunk computes at level `k`, the second chunk computes at level `k-1`, and so on. Please see Figures 6.4-6.5 in the textbook for illustrations of such wavefronts.

3 The assignment

You are requested to parallelize the Guass-Seidel algorithm using two chunks, so that two MPI processes can divide the work on each wavefront. (Ambitious students are welcome to extend the parallelization to more chunks.)

That is, the computational points in the j-direction (`j=1,2,...,jmax-2`) are to be divided into two chunks, with the left chunk covering indices `j=1,2,...,jmax/2-1` and the right chunk being responsible for `j=jmax/2,jmax/2+1,...,jmax-2`. Since two chunks are used, there will be in total `kmax-1` wave fronts. Except for the first and last wavefronts, each wavefront can be computed by two MPI processes in parallel, with process 0 working on the left chunk at level `k`, and process 1 working on the right chunk at level `k-1`.

It is important to remember that inter-process communication is needed. When process 0 finishes computing at level `k`, its computed right-most values, `phi[k][jmax/2-1][i]`, for `i=1,2,...,imax-2`, should be sent to process 1. In return, when process 1 finishes computing at level `k-1`, its computed left-most values should be sent to process 0. Note that communication is also needed on the first and last wavefronts, although only one process has computation to do.

3.1 Serial programming tasks

You are requested to first implement a serial version of the Gauss-Seidel iteration that is based on two-chunk traversal of the wavefronts, using the following code skeleton:

```
void GS_iteration_2_chunks (int kmax, int jmax, int imax, double ***phi)
{
  // first wavefront: only computation on left chunk at level k=1
  // ...

  for (k=2; k<=kmax-2; k++) {
     // computation on left chunk at level k
     // ...
     // computation on right chunk at level k-1
     // ...
  }

  // last wavefront: only computation on right chunk at level k=kmax-2
  // ...
}
```

This function is meant to run serially (no parallelization or communication), with the purpose of familiarizing yourself with traversal based on wavefronts, and verifying that `GS_iteration_normal` and `GS_iteration_2_chunks` produce identical computational results. In addition, you are requested to implement two serial help functions:

- `void allocate_array3D (int kmax, int jmax, int imax, double ****array)` Note: The input argument named `array` is of type `double****` (quadruple pointer), because a 3D array is to be allocated inside the function.
- `double euclidean_distance (int kmax, int jmax, int imax, double ***arr1, double ***arr2)` This function can be used to check whether `GS_iteration_2_chunks` indeed produces the identical computational result as `GS_iteration_normal`.

Finally, you are requested to implement a serial main function with the following structure:

```
int main (int nargs, char **args)
{
  double ***arr1, ***arr2;
  int n,k,j,i, num_iters, kmax, jmax, imax;

  // read from command line the values of num_iters, kmax, jmax, and imax
```

```
  // ...

  allocate_array3D(kmax,jmax,imax,&arr1);
  allocate_array3D(kmax,jmax,imax,&arr2);

  // assign same initial values to arr1 & arr2 (don't give a constant to all entries of arr1 & arr2)
  // ....

  for (n=0; n<num_iters; n++) {
    GS_iteration_normal(kmax,jmax,imax,arr1);
    GS_iteration_2_chunks(kmax,jmax,imax,arr2);
  }

  printf("num iters=%d, kmax=%d, jmax=%d, imax=%d, diff=%g\n",
         num_iters, kmax, jmax, imax, euclidean_distance(kmax,jmax,imax,arr1,arr2));

  return 0;
}
```

3.2 Parallel programming

You are requested to implement the following function

```
void GS_iteration_2_chunks_mpi (int my_rank, int kmax, int my_jmax, int imax, double ***my_phi)
```

Different from the serial counterpart `GS_iteration_2_chunks`, this MPI version has one additional input argument `my_rank` that has value either 0 or 1. That is, `GS_iteration_2_chunks_mpi` is to be called by MPI process with rank 0 or 1. Moreover, the input 3D array `my_phi` per MPI process is of size `kmax`×`my_jmax`×`imax`, with the value of `my_jmax` equal `jmax/2+1` (assuming `jmax` is divisible by 2). This is because each process is assigned with `jmax/2-1` computational points in the `j`-direction, plus one end-point being a boundary point and the other end-point being a ghost point (needed for receiving data from the neighbor process). **Remember to implement the needed inter-process communication after the computation is done on each wavefront.**

You are also requested to implement an MPI version of the `main` function, which lets process 0 read values of `num_iters`, `kmax`, `jmax` and `imax` from the command line. These are then broadcasted to process 1. Thereafter, each process allocates a 3D array named `my_arr` of size `kmax`×`my_jmax`×`imax`, with `my_jmax` equal `jmax/2+1`. After assigning suitable initial values to `my_arr`, both processes call `num_iters` iterations of `GS_iteration_2_chunks_mpi`. Thereafter, process 1 sends the content of its computed 3D array `my_arr` to process 0, which can thereby construct a global 3D array of size `kmax`×`jmax`×`imax`. Finally, process 0 can compare this global 3D array with the computational result produced by running the serial function `GS_iteration_2_chunks` (using the same initial values and number of iterations) for validating the correctness of the MPI implementation.

3.3 Submission

Each student should submit, via Devilry, a tarball (`.tar`) or a zip file (`.zip`). Upon unpacking/unzipping it should produce a folder named `IN3200_Oblig2_xxx` or `IN4200_Oblig2_xxx`, where `xxx` should be the username of the student. There should be two sub-folders, one containing the serial code, the other containing the MPI code. (Each function should be placed in a separate file. Ideally, there should be a Makefile in each sub-folder.) In addition, there should be a `README.txt` or `README.md` file that contains the most essential information about compilation and an example of how to run the two main programs.