High-Performance Computing
Project : parallel Gauss-Seidel method

Student: Horace GUY — Report on project, IS1260 2017

1 Introduction

The assignment was to implement the Gauss-Seidel method in parallel. It aims to achieve the resolution of a linear system of the form :

$$Ax = b$$

where A, in the code noted `mat` or `matrix`, is a square matrix, and b the parameters vector denoted by `param`. The scheme is to iterate a set of operations, from an initial vector `init` until convergence at the precision level `epsilon` desired.

The method is guaranteed to converge for strictly dominant diagonal matrixes, thus I only used these kind of samples to test the program.

I closely followed the method described by Yueqiang Shang, Faculty of Science, Xi'an Jiaotong University, published in Elsevier, section Computers and Mathematics with Applications 57 (2009) 1369 _ 1376. The article is available at `http://www.sciencedirect.com/science/article/pii/S089812210900042X`.

In regard to simplicity, and because of a lack of time, I only implemented the method for the case where the parameter g, the height of the blocks that split the matrix and vectors, satisfies the following : *g divides dim*, the dimension of the matrix and vector.

2 Structure of the program

According to the reference document, the subalgorithms followed by the processors are as follow :

master processor

— Check that the values given are compatible (g divides dim).

— Load data from the text files init.txt, param.txt, mat.txt.

— Broadcast `init` and `param` vector to the slaves.

— scatter the matrix to the slaves (including p0), with the procedure `scatter_all()`. This procedure uses a tensor `blocs_feeder[][][]` to partition the matrix in blocs of size `p*g`.

— At the end of each iteration in the `while ((k < max_iter)and(sign == 0)) {...}` , receive the result `x_new[]` from processor `last`, overwrite it in `x_pred[]` and broadcast it to slaves in the same buffer.

— If algorithm converged with precision `epsilon` or `max_iter` is reached, print the result and write it in the document "final.txt" if converged.

— Otherwise, begin next iteration.

slave processor number j >= 0

— Receive `init, param` vectors from p0's broadcast.

— Receive the data `own_blocs[][][]` from p0's scatter of the matrix.

— For `k = 0, ..., max_iter` or until convergence :

- set sign = 1, compute $t_i^{(k)}$ values for appropriated is, with `compute_ts()` , stored in the `T[][]` array. Launch `init_zs()` which initializes the values of $z_i^{(k+1)}$ at 0 in the `Z[][]` array, for values of i in the range of proc. j.
- For qb = 0, ..., `own_blocs_count -1 :`
  - receive sign in the buffer `sign` and $x_0^{(k+1)}, ...x_{qb*p*g+j*g-1}^{(k+1)}$ in the buffer `x_new[]` from (j-1)%p -th processor.
  - compute $x_{qb*p*g+j*g}^{(k+1)}...x_{qb*p*g+(j+1)*g}^{(k+1)}$ and complete computation of corresponding values of $z_i^{(k+1)}$ with the procedure `compute_missing_z()`; check if requirement of precision is reached - if not, set sign = 0.
  - send $x_0^{(k+1)}...x_{qb*p*g+(j+1)*g}^{(k+1)}$ and sign to (j+1)%p-th processor
  - compute portions of partial $z_i^{(k+1)}$ sums, involving the items freshly collected and computed, with `update_zs()`. The number of such relevant values of $x_i^{(k+1)}$ is p*g if `qb == 0`, and `my_rank*g` otherwise.
- If `my_rank == last`, that means that I am the one ending the iteration ; i.e. I have the complete `x_new[]` vector; I send it to p0.
- I receive the broadcast of `x_new[]` performed by p0, and stores it in my own `x_pred[]` buffer.

3 Results

To try the resolution method, I wrote a "create_samples.c" program that provides the user with strictly dominant diagonal matrixes, random param vector and zero init vector. One has to run the .exe application and choose the value of `dim` to create new arrays. Then one has to copy and paste the files "mat.txt", "init.txt" and "param.txt" in the parent folder to run the resolution algorithm.

I have tried the program with an initialisation at zero vector, random parameter b with values in the interval [0,dim]. The matrixes are the ones used by M. Shang in the articles.

After a lot of debugging, the algorithm seems to work fine whenever g divides dim. Even in the case of p not dividing dim/g, the program terminates and often converges quickly. One has to change manually the parameters such as `g, epsilon, max_iter` at the beginning of the `main()` function.

I provided an example of size 12 in this folder, in order to print relevant informations without flooding the command line screen.

The major lead of improvements are first the case when g doesn't divide dim. Also, the arrays `float matrix[][]` and `float blocs_feeder[][][]` are defined in every processor, such that each local memory allocates two times a `float dim * dim` space, and then doesn't use it. This could be fixed, but some tricky manipulations of pointers are required and I didn't manage to implement it very well when I tried.