PARALLEL PROGRAMMING

UPMC - SORBONNE UNIVERSITÉ

Parallel Programming

Spherical Harmonic Representation of Earth Elevation Data

Authors:
21217533 Marco PARADINA
21212087 Vignesh ANANTHARAMAKRISHNAN
3808688 Diego ARBULU SEDANO

Course Lead:
Charles Bouillaguet

November 28, 2022

1 Introduction

Among the different methods available to improve the computation, we chose to implement Scalapck's library functions for the least-square problem solving and distribution among processors. Specifically pdgeadd and pdgels for the optimized linear solving methods and the Blacs grid functions for the parallelism. Simulteanously we rewrote the load_data_points function in harmonics.c to improve the model's accuracy.

1.1 Objectives

- Implementation of Scalapack's "pdgels"
- Implementation of Blacs' "gridinit"
- Rewriting of load_data_points()

2 Implementation

2.1 Changed Functions

In the harmonics.c file, we changed the load_data_points() function that was used to pick N points on the input datafile. This was done as, intuitively, it was much more logical to pick N equally spaced points instead of the first N points, as to not over specialize the model on region-specific data. The resulting models had a better accuracy when tested with the validate function. This was more true for smaller data inputs, as when n is equal to the total points available in the file, there's virtually no difference between the first algorithm and our rewritten one.

The bulk of our approach to solving this project however stems from altering the linear_least_squares function where we attempted to use Scalapack's pdgels_ function to solve the linear least squares problem in parallel time which had an OpenMPI backend to handle the block-cyclic distribution of the matrices A and b.

2.2 Added Functions (ScaLAPACK implementation)

To distribute the least-square problem we used the ScaLAPACK library. The following conventions will be used to describe the implementation:

- The A matrix is the one defined in model.c
- The matrix b is the vetor defined in model.c as data.V

The least-square problem is then to minimize $|Ax - b|$. To do this we first distributed the matrix A among all the processes (which are arranged in a process grid). This way the global matrix A is only owned by the first process, i.e. the one with coordinates $(0, 0)$ in the process grid, and each process owns a local smaller matrix A_{distr}. The distribution of A is performed calling the **pdgemr2d** ScaLAPACK routine.

The same is done for b: the global array b is owned by process $(0, 0)$ and distributed into smaller b_{distr} arrays.

Then, to compute the actual least-square-solution, the **pdgels** routine is called. This computes the local least-square-solution to $|A_{distr}x - b_{distr}|$. As can be seen in the code, two calls are necessary: in the first one the argument **lwork** is set to -1, so that the call is seen as a query to find the optimal workspace size, i.e. the optimal size for the **work** array. The second call actually computes the least-square-solution and stores it in b_{distr}.

Then all these local solutions must be gathered to obtain the global least-square solution. This is done with another call to **pdgemr2d**, which gathers all the b_{distr} into b. It is not necessary to gather A as well because the least-square-solution is stored in b.

When very large matrices are involved, some other parameters become very large as well and cannot be be represented by an int type anymore. Because of this, all the variables that in the original sequential code were declared as int are now declared as long.

The **scalapack.h** header file contains all the signatures of ScaLAPACK and Cblacs routines. These are linked to the **pmodel.c** file, where the actual calls to those routines happen, using the flags that can be found in the Makefile.

2.3 Difficulties Encountered

The most challenging aspect of using ScaLAPACK routines was the quality and availability of documentation: the description of the arguments of the routines used here are often unclear or very ambiguous. Examples of implementations found online are very rare and usually not well written or correct, hence not helpful. This resulted in spending a lot of time trying different interpretations of the documentation to find the correct one.

2.4 Limitations and possible improvements

The main limit of this implementation is that the entire A matrix must be stored in one of the processes. This limits the possible size of A, and therefore the precision of the data that we are computing, since the size of A is determined by the number of observed points in the dataset and by $lmax$. To improve this, the A matrix should never be created. Each process should read the input file and create its local matrix A_{distr}, instead of the $(0, 0)$ process creating A for it to be distributed by **pdgemr2d**. The memory constraint would then be the size of the local matrices A_{distr}, which is much less restrictive than the size of A.

3 Performance

3.1 Accuracy

The first modification added was the rewriting of the load_data_points(), as it works on the sequential code. There were no performance enhancement but the accuracy decreased by an order of eˆ8. As seen on the fig. below with the computation on 10000 points with an lmax of 10 :

```
diego@diego-VirtualBox:~/Documents/PPAR/Project$ ./model --data ETOPO1_small.csv --model model.txt --npoint 10000 --lmax 10
Linear Least Squares with dimension 10000 x 121
Matrix size: 9.7 MB
Reading data points from ETOPO1_small.csv
Successfully read 10000 data points
Building matrix
Least Squares (292.8 MFLOP)
Completed in 0.4 s (678.2 M FLOPS)
residual sum of squares 5.01182e+09
Saving model in model.txt
diego@diego-VirtualBox:~/Documents/PPAR/Project$ ./validate --data ETOPO1_small.csv --model model.txt --lmax 10
Reading order-10 model from model.txt
Successfully read model
Reading data points from ETOPO1_small.csv
Completed in 0.1 s
Average error 4.25834e+08 meters
Max error 5.20876e+09 meters
Standard Deviation 1.01125e+09
```

fig.1 - The sequential algorithm with the initially chosen points

```
diego@diego-VirtualBox:~/Documents/PPAR_project_1$ ./model_old --data ETOPO1_small.csv --model model_old.txt --npoint 10000 --lmax 10
Linear Least Squares with dimension 10000 x 121
Matrix size: 9.7 MB
Reading data points from ETOPO1_small.csv
Successfully read 10000 data points
Building matrix
Least Squares (292.8 MFLOP)
Completed in 0.4 s (664.4 M FLOPS)
residual sum of squares 1.37116e+10
Saving model in model_old.txt
diego@diego-VirtualBox:~/Documents/PPAR_project_1$ ./validate --data ETOPO1_small.csv --model model_old.txt --lmax 10
Reading order-10 model from model_old.txt
Successfully read model
Reading data points from ETOPO1_small.csv
Completed in 0.3 s
Average error 924.506 meters
Max error 6955.29 meters
Standard Deviation 1211.26
```

fig.2 - The sequential algorithm with equally spaced points

3.2 Computation Time

In the parallel code, the performance is evaluated on different settings. The most notable enhancement is in the comparison of the computation from the sequential algorithm to the parallel algorithm run on 8 processors. Both loaded 100000 points and had an lmax of 48.

```
darbulusedano@grisou-18:~$ ./model_old --data ETOPO1_high.csv --model model_old.txt --npoint 100
000 --lmax 48
Linear Least Squares with dimension 100000 x 2401
Matrix size: 1.9 GB
Reading data points from ETOPO1_high.csv
nmax/npoint = step : 6480000 / 100000 = 64
Successfully read 100000 data points
Building matrix
Least Squares (1.2 TFLOP)
Completed in 1911.3 s (603.2 M FLOPS)
residual sum of squares 3.14574e+10
Saving model in model_old.txt
```

fig.3 - The sequential algorithm with said settings

```
nmax/npoint = step : 6480000 / 100000 = 64
Completed in 22.2 s
residual sum of squares 3.14574e+10
Saving model in out.csv
```

fig.4 - The parallel algorithm with the same settings

In addition to that, different performance evaluations were done on 4, 8 and 16 processors, with varying overall parameters. Each performance table has an added line of the sequential program computation for comparison purposes :

- npoint: 64800

Table 1: 4 processes with lmax 24

Nb proc.	Block param.	Time (in s)
1	1	84.7
4	1	12
4	2	10
4	3	9.7
4	4	9.2
4	5	8.4
4	6	8.2
4	7	8.3
4	8	8.6
4	9	8.9
4	10	10.2
4	25	13.9
4	100	16.4

Table 2: 8 processes with lmax 48

Nb proc.	Block param.	Time (in s)
1	1	108.8
8	1	108.1
8	2	61.9
8	3	44.8
8	4	36.1
8	5	30.9
8	6	25.8
8	10	18
8	25	17.8
8	100	17.3

Table 3: 16 processes with lmax 48

Nb proc.	Block param.	Time (in s)
1	1	108.8
16	1	68.5
16	2	37.5
16	3	27.3
16	5	18.8
16	6	16.1
16	7	15.7
16	8	13.5
16	9	13.9
16	10	12.3
16	25	11.8
16	100	14.4

Overall these computations behave similarly, the computation time decreases as the block repartition gets bigger, they stabilize around a certain setting and then the computation increases again, probably because the processors have to store more and further apart in physical memory.
On table 1 the biggest speedup can be evaluated as 84.7/8.2 = 10.3, with blocking parameter 6. Which accounts for an efficiency of 10.3/4 = 2.6 per process
For table 2 it's 108.8/17.3 = 6.3, with blocking parameter 100. And that's an efficiency of 6.3/8 = 0.78 per process
For table 3 it's 108.8/11.8 = 9.2, with blocking parameter 25. And that's an efficiency of 9.2/16 = 0.58 per process

- npoint: 100000

Table 4: 4 processes with lmax 24

Nb proc.	Block param.	Time (in s)
1	1	159
4	1	22.2
4	2	13
4	3	9.5
4	4	8
4	5	6.9
4	6	5.9
4	7	5.7
4	8	5.1
4	9	5.2
4	10	4.5
4	15	4.7
4	17	4.7
4	19	4.6
4	20	4
4	21	4.6
4	25	4.9
4	100	6.7

Table 5: 8 processors with lmax 48

Nb proc.	Block param.	Time (in s)
1	1	1911.3
8	1	169
8	2	95.4
8	3	69.8
8	4	56.3
8	5	48.1
8	6	40.6
8	7	39.3
8	8	32.1
8	9	34.8
8	10	28.2
8	15	29.9
8	17	29
8	19	28.6
8	20	22.3
8	21	29
8	25	27.7
8	100	27.5

On table 4, the biggest speedup is 159/4 = 39.75, with blocking parameter 20. That's an efficiency of 39.75/4 = 9.94 per process
For table 5 it's 1911.3/22.3 = 85.7, with blocking parameter 20. And that's an efficiency of 85.7/8 = 10.71 per process

4 Results

This section describes the computation performed with a large number of cores on g5k to obtain the most precise solution possible for the least-square-problem.
The computation was performed with the following parameters:

- npoint: 120000
- lmax: 124
- input file: ETOPO1_ultra.csv
- cluster used: gros (nancy)
- number of nodes: 54
- number of processes: 961
- process grid dimensions: 31x31
- blocking parameters: all set to 50

The following command was used to reserve the g5k resources for the computation:

oarsub -p gros -l host=54,walltime=t -r '2022-11-dd hh:mm:ss'

To run the computation on g5k, the ScaLAPACK library needs to be installed in all the nodes that are being used for the computation. This is done by the **install_scalapack.sh** bash script, which iterates through all the nodes found in OAR_NODEFILE to install the library.

Then the the computation is launched with the following command:

mpirun -n 961 -machinefile $OAR_NODEFILE ./pmodel /
–data /srv/storage/ppar@storage1.nancy.grid5000.fr/ETOPO1_ultra.csv –model out.csv –npoint 120000 –lmax 124

Finally the result is validated with the following command:

./validate –data /srv/storage/ppar@storage1.nancy.grid5000.fr/ETOPO1_ultra.csv –model out.csv –lmax 124

These are the results obtained after validation:

AVG ERROR, STD DEVIATION ETC. WILL BE PASTED HERE

All further details about compilation and executions can be found in the README.