Sherical Harmonic Representation of Earth Elevation Data - Project Report

Auguste Warmé-Janville (3800191)

November 25, 2022

1 Introduction

To acknowledge which part of the sequential code needed to be parallelized, i used a GNU profiler to determine which functions were consuming the most ressources. It turned out that the QR factorization was by far the most time consuming function call. That is the part of the code we are going to try parallelize.

2 Parallelism strategy

I chose to use the ScaLAPACK library to parallelize the QR decomposition. The matrix needs to be distributed between all the processes in a 2D block cyclic distribution for the factorization to work. Then it is sent back to the main process for the least squares problem to be minimised. All the communication between the processes are done using BLACS or MPI routines.
After scaling the computations to the final size, it turned out that I had to parallelize the validate function as well, since it was taking more time to validate the models than to compute them. The parallelism here is simple, each process checks one line every N line (where N is the total number of processes). For all the lines to be read, each process starts reading with an offset equal to its `rank`. It is done using MPI send and receive routines.

ScaLAPACK

ScaLAPACK is a parallel version of the LAPACK library. It provides high performance linear algrebra parallel routines.

BLACS

BLACS provides an linear algrebra oriented message passing interface based on MPI.

3 Code : ScaLAPACK and BLAS routines used

The following ScaLAPACK routines were used to make the parallel QR factorization work. The functions descriptions are extracts from the ScaLAPACK documentation.

```
/*
* This function computes the local number of rows or columns of a
* block-cyclically distributed matrix contained in a process row or
* process columm, respectively, indicated by the calling sequence argument iproc.
*/
long numroc_(long* N, long* NB, long* IPROC, long *ISRCPROC, long* NPROCS );

/*
* This function computes the process row or column index of a
* global element of a block-cyclically distributed matrix.
*/
long indxg2p_( long* INDXGLOB, long* NB, long* IPROC, long* ISRCPROC, long* NPROCS )
    ;

/*
* The pdgemr2d routine copies the indicated matrix or submatrix
* of A to the indicated matrix or submatrix of B. It provides a
* truly general copy from any block cyclicly-distributed matrix
* or submatrix to any other block cyclicly-distributed matrix or submatrix.
*/
void pdgemr2d_( long* m, long* n, double* a, long* ia, long* ja, long* desca, double
    * b, long* ib, long* jb, long* descb, long* ictxt);
```

```
/*
* These subroutines compute the QR factorization of a general matrix A,
* where, in this description:
* A represents the global general submatrix Aia:ia+m-1, ja:ja+n-1 to be factored.
* For PDGEQRF, Q is an orthogonal matrix.
* For m >= n, R is an upper triangular matrix.
* If m = 0 or n = 0, no computation is performed and the subroutine returns
* after doing some parameter checking.
*/
void pdgeqrf_(long* M, long *N, double* A, long* IA, long *JA, long* DESCA, double *
    TAU, double *WORK, long* LWORK, long *INFO);

/*
* This subroutine initializes a type-1 array descriptor with error checking
*/
void descinit_ (long *desc, const long *m, const long *n, const long *mb, const long
     *nb, const long *irsrc, const long *icsrc, const long *ictxt, const long *lld,
    long *info);
```

The BLACS routines used are the following.

```
extern void Cblacs_get(long icontxt, long what, long *val);
extern void Cblacs_setup(long * rank, long * nprocs);
extern void Cblacs_gridinit(long * ictx, const char * order, long nprow, long npcol)
    ;
extern void Cblacs_gridinfo(long ictx, long * nprow, long * npcol, long * myrow,
    long * mycol);
extern void Cblacs_gridexit( long ictx );
extern void Cblacs_exit( long doneflag );
```

4 Implementation comments

4.1 Process grid

For the ScaLAPACK functions to work, processes need to be arranged in a process grid [1]. This is done using BLACS functions. As mentioned in the ScaLAPACK documenation, the process grid need to be a square in order to get the best perfoemance.

4.2 Matrices 2D block cyclic distribution

The matrices need to be distributed between the processes following a 2D block cyclic distribution [2] for the factorization to work. It is acheived using the `pdgemr2d` function.
The dimensions of the process grid p and q are computed by the `approx_sqrt(int a, int *p, int *q)` function which computes the integers p and q ($p \leq q$) such that $a = p \times q$ and the length of the interval $[p, q]$ is minimal. If a is a square, $p = q = \sqrt{a}$. The size of the blocks is determined by the global variable `BLOCK_SIZE`. The ScaLAPACK user guide recommends a block size of 64 [3].

4.3 Points selection

In order to reduce the size of the matrices we're working on, we need to skip some datapoints on the hi and ultra sets. Thus, I modified the function that reads the points from the file and added a command line option to the *model* program : `--spacing`. It is quite straightforward. If a spacing of N points is specified, the program will sample only 1 point every N points from the dataset. The `--npoint` option remains unchanged. It must be set to the max number of points the set contains for the program to iterate over all of them. As a result, the value of the variable `npoint` from the main function, which represents the number of rows of the matrix (which is different from the command line option) is equal to *npoints/spacing*.

4.4 Running the code on multiple g5k nodes : HOW2

The main focus here is that ScaLAPACK needs to be installed on each session on every available node before running the `model` function. These are the steps to follow :

- Reserve some nodes from the same cluster
- Install scalapack on all the nodes via ssh : run the `install_scalapack.sh` script

[1] *About process grids*
[2] *About 2D block cyclic distribution*
[3] *About ScaLAPACK performance*

- Compile the code (`make`)
- Run it
  - model :

```
mpirun -n <core_count> -machinefile $OAR_NODEFILE ./model --data <filename>
--model <model> --npoint <npoints> --lmax <lmax> --spacing <spacing>
```

  - validate :

```
mpirun -n <core_count> -machinefile $OAR_NODEFILE ./validate --data <filename>
--model <model> --lmax <lmax>
```

5 Speedup, Scalability

5.1 Large Matrix : 2.1Go size, 4:1 ratio

These results were acheived using 4 nodes (64 cores) from the grisou cluster. We're measuring the performance of the implementation on large matrices. We're also comparing the performance depending on the block size.

Model, block_size=8

Processor count	Computing time (in seconds)	Speedup	Efficiency	GFLOPS
1	1592	1	1	2.7
8	104.25	15.27	1.91	41
16	63.95	24.89	1.56	66
24	46.25	34.42	1.43	91
32	37.33	42.65	1.33	114
40	32.34	49.23	1.23	131
48	28.61	55.64	1.16	149
56	26.32	60.49	1.08	162
64	24.43	65.17	1.02	174

Table 1: *Model : npoint = 32400, nvar = 8100, hi dataset, spacing = 200, block size = 8*

Model, block_size=64

Processor count	Computing time (in seconds)	Speedup	Efficiency	GFLOPS
1	1574	1	1	2.7
8	62.78	25.07	3.13	68
16	37.17	42.35	2.65	114
24	29.25	53.81	2.24	145
32	25.29	62.24	1.95	168
40	22.42	70.21	1.76	189
48	20.38	77.23	1.61	208
56	20.19	77.96	1.39	210
64	19.07	82.54	1.29	223

Table 2: *Model : npoint = 32400, nvar = 8100, hi dataset, spacing = 200, block size = 64*

Validate

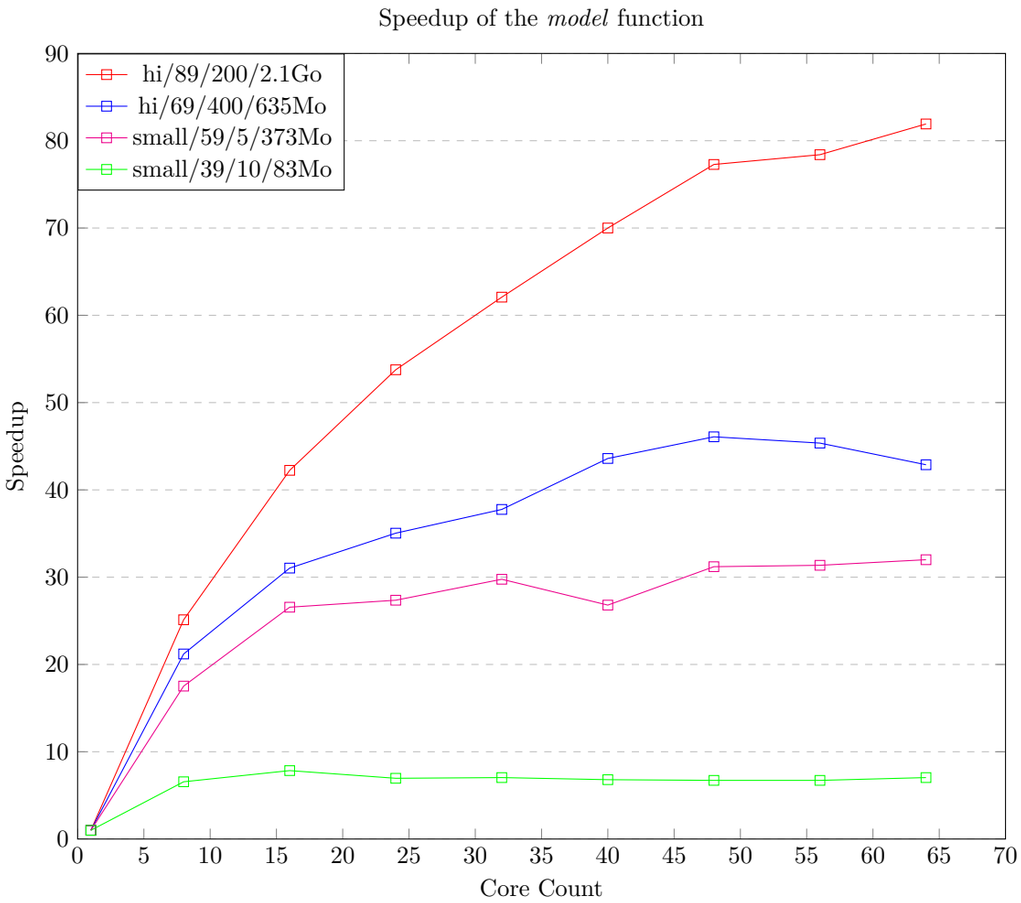
Processor count	Computing time (in seconds)	Speedup	Efficiency
1	707	1	1
8	98.9	7.15	0.89
16	59.9	11.8	0.74
24	38.3	18.46	0.77
32	30.1	23.49	0.73
40	23.5	30.09	0.75
48	18.8	37.61	0.78
56	16.8	42.08	0.75
64	14.6	48.42	0.76

Table 3: *Validate : lmax = 89, hi dataset*

As specified in the ScaLAPACK user manual, the efficiency of this computation is greater with a block size of 64 compared to a block size of 8. Indeed, as the block size increases, the number of communication routines called decreases, thus the global computation runtime also decreases.
As well as the speedup induced by the pallarelization, the parralel ScaLAPACK version is overall more efficient than the sequential program, as the speedup is superior to 1 for any core count greater than one on the `model` function analysis. This makes ScaLAPACK a good choice for big linear algebra computations (what a surpise !).

5.2 Comparison between different matrices sizes

I measured the speedup of the model function for different matrices dimensions (4:1 ratio) on different sets, and plotted the results. The core count varies from 8 to 64 by increments of 8. The legend follows the formatting *dataset/lmax/spacing/matrix_size*. The block size is equal to 64.



As the size of the matrix decreases, the speedup incuded by the parallelization also decreases. To a reduced scale of time, the inter-processes communications takes a not negligible amount of time. If the parallel `model` program is run on too small matrices, the performance gain becomes null. Therefore, it is not especially a good idea to run the parallel program on small matrices, it is just going to consume more energy, if it doesn't simply makes the sequential program slower. An idea to improve the sequential performance would the be to use LAPACK functions that are designed for these kinds of computations.

5.3 Conclusion

Parallelizing the sequential code using ScaLAPACK turns out to be very efficient, as long as the matrices on which the computation is done are big enough. However, this implementation can not be scaled to matrices with sizes greater than 16 Go (see the section about Bugs and Crashes).

6 Best model

The best model I computed is contained in the file `model_ul_129_2200.txt`. The parameters are :

- dataset = ultra
- lmax = 129
- npoints = 233280000
- spacing = 2200

Running this program produces a matrix which dimensions are 106036 x 16900. It requires 14.3 GB RAM.
The `validate` function returns :

- Average error = 229.034 meters
- Max error = 6781.6 meters

- Standard Deviation = 382.026

Since it looks like working with matrices with size greater than 16Go makes the program crash, I could not push the computation any further.

7 Bugs/Crashes

On large matrices, the code often crashes for certain values of lmax (`xxmr2d:out of memory`). This has to do with the 2D block cyclic distribution function. As well, on very large matrices (16 Go size), the program crashes (`segfault, address not mapped`). This is caused by the ScaLAPACK prebuilt libary which uses 4 bytes integers, thus the max size for a double precision float matrix is 16 Go. A potential fix would be to rebuild the library with 8 bytes integers, but I did not had time to do it.