|epcc|

EPCC-SS98-9

Optimisation of parallel vector-topological data input for Geographical Information Systems using MPI-IO

Eurico Moita

Abstract

GIS processing is both computationally and I/O intensive, making it an ideal application for parallel processing. A parallel library of core GIS operations has been designed and partly implemented. The performance of most of these operations depends on the vector input data module, which in it's current implementation suffers from poor scalability. This paper investigates the pontential use of MPI-IO in improving the performance of this module.

Contents

1 Introduction

1.1 Geographical Information Systems (GIS)

There has been an increasing use of GIS packages to solve problems related to geographic distribution. As the operations are very complex and usually carried out on very large datasets their suitability for parallelisation is much desired.

A parallel library of some of these operations has been designed and partly implemented, forming an interface to some major commercial GIS packages. Scalability of course, is of paramount importance.

One part of this design, the vector input data module, is a corner stone of most of the mentioned operations, performing the required pre-processing (usually under the form of sorting) and distribution of data. The current implementation of this module suffers from poor scalability, caused by the bottleneck created by the use of non parallel I/O [1].

1.2 MPI-IO

MPI-IO, a part of MPI-2, has been designed to address the major problems encountered in parallel I/O [4]. It defines an interface that allows processes to define their file access patterns and make collective I/O calls, two features that play an important role in obtaining high performance parallel I/O.

2 Background

2.1 Overview of MPI-IO

2.1.1 Motivation

In general, many parallel applications access small non-contiguous portions of a file concurrently [4].

These "scattered-type" access patterns, tend to form a bottleneck due to the overhead incurred in having several processes seeking through shared files back and forth to read or write small chunks of data. An increase in the number of processes aggravates the problem further, inhibiting the desired scalability. In order to increase I/O performance, some optimizations must be done on the actual physical access to the storage media. A proposed manner of doing this is to collect several small non-contiguous requests for data into one large contiguous block.

MPI-IO, the I/O part of MPI-2, defines an interface for parallel file access. It also aims to provide improved efficiency through a high-level interface that allows users to specify requests to several non-contiguous blocks of data in one function call, by defining the access patterns of the process.

2.1.2 Optimizations

Armed with this information, the various implementations of MPI-IO can optimize data transfer to and from files, minimizing the problems mentioned above. In particular ROMIO, a portable MPI-IO implementation, performs at least two kinds of optimization: data sieving and collective I/O.

Data sieving is used to avoid I/O latency by reducing the number of requests to the file system. The implementation performs a few large contiguous requests to the file system even when the actual call is for several small non-contiguous accesses. The actual data is then taken from the buffer used to store the large block, which although containing more information than was actually requested, improves the accesses by reducing the number of requests to the file systems, as previously stated.

Collective I/O is used to group several requests from various processes onto a smaller number of file system requests, by trying to merge the several non-contiguous accesses onto a small number of contiguous ones (where data sieving is used if the resulting merged accesses are not contiguous in themselves).

The process happens in two phases, an I/O phase during which the data is read to a buffer and a communication phase when the several pieces of data are distributed to the respective processes. There is some overhead in the communication phase, but it's cost is negligible.

To benefit from this optimization, all processes that together opened the file must use collective-I/O function calls.

2.1.3 Classification of I/O requests

MPI-IO classifies I/O access patterns into *levels 0* through to *3*. *Level 0* refers to UNIX style access in which one request is made per piece of data.

Level 1 refers to the same type of request but done collectively, so that the implementation can try to optimize access knowing that each process will use the same call, by attempting a merged access.

Using *level 2*, each request is made using a specified non-contiguous access pattern, ie, the function call is given information on how the wanted information is disposed on the file, and so tries to use data sieving. Here's an example:

Imagine a process accessing data on a file in the following pattern (where each dot indicates a blank and 1 indicates data):11 . . . . 11 . . . . 11 11

If this was to be done using *level 0* (ie, no information is given on the access pattern) then for each piece of data there would be one filesystem request adding I/O latency. If, however, the function call knows the above pattern then it might try to optimize by reading larger chunks of the file at a time and then passing onto the calling code the relevant information, thus reducing the amount of filesystem requests.

Finally *level 3* is a combination of *level 1* and *2* where each process gives information on it's access pattern and uses collective I/O. As an example consider three processes with the following access pattern (same notation as before with the addition of the symbols '2' and '3' to indicate the data's owner):1122 . 3 . 1122 . 3 . 1122 . 3, notice how each process has 3 pieces of it's data distributed on the file. Again *level 0* would result in large amount of requests (in the region of 9). *Level 1* might optimize into 3 file requests(one for each piece of data each process owns on the file) as the requests from different processes are merged into one. *Level 2* might again optimize into 3 file request (this time one per each process as the call tries to read all the data specified in the access pattern). And given a *level 3* type of information, the function call might try to merge all requests into one access, knowing the access pattern for each process and also that they all want to read the interleaved data. Some optimization is also possible if the data is not interleaved. So, the higher the *level*, the higher the content of information passed onto the function call, the better a chance for optimization. Therefore users are advised to use *level 3* when possible.

2.2 Geographical Information Systems - GIS

Geographical Information Systems permit the storage, display and manipulation of spatially referenced data. Continued rapid growth in the availability of digital cartographic data and satellite images is creating a demand for intensive computing to integrate and process large datasets. Such datasets can be collected and integrated from a wide variety of sources, including geological, political and census databases. This information can then be used to solve problems in areas such as environmental assessment, transportation, marketing, distribution, telecommunications, planning and resource management.

2.2.1 GIS Data

Within a GIS, geographical features are represented by their spatial attributes (eg. extent and location) and non-spatial ones that classify the type of the area (eg. terrain, vegetation, etc)[2]. There are two kinds of representation for geographical features: raster, in which an area is represented using a matrix of cells each with one or more attributes, and vector-topological where each feature is represented using geometrical objects (eg. points, lines and polygons). One of the particular features of the latter format, is the separation between the feature's spatial information and it's other attributes, where unique identifiers link these two kinds of data in a relational way. The operations on this type of data need to follow several links between records to gather the needed information, that sometimes must be sorted in a suitable way to the algorithm implementing the operation. This task of gathering and sorting the required data is usually very intensive, both CPU and I/O wise.

2.2.2 The vector-topological data input module

This module performs the pre-processing and distribution of vector data necessary to carry out the desired operations in parallel. It's design consists of a *Source* process that acts as a file server and co-ordinator for several *Pool* processes that actually perform the operations [1]. This structure creates a severe bottleneck, as all global file accesses must go through the source process. This is shown in figures 1 and 2.

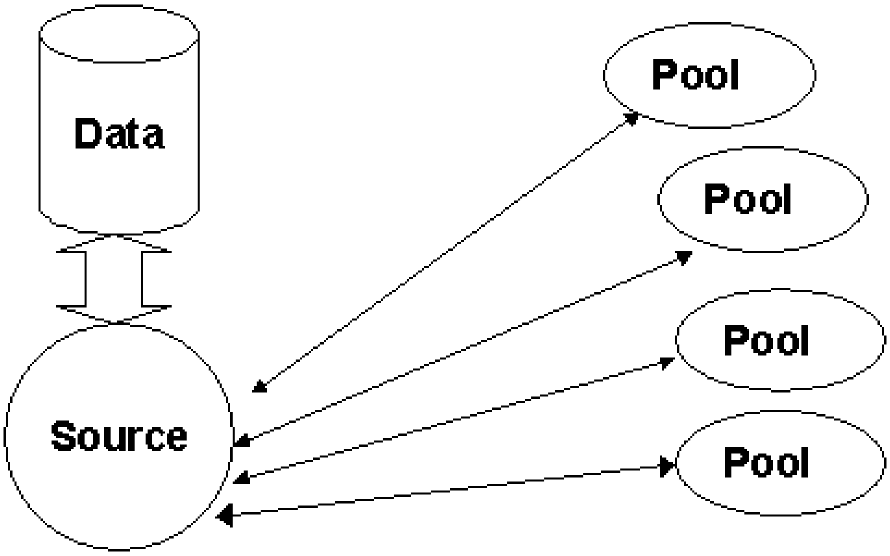

Figure 1: information flow in the current design

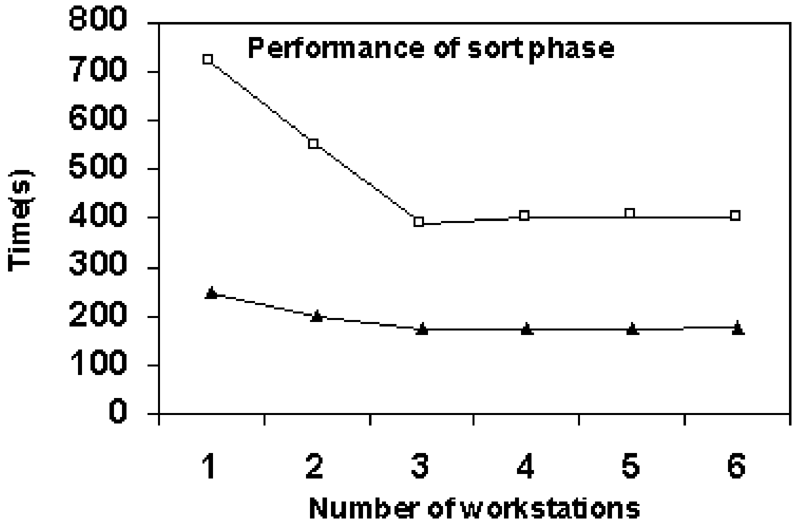

Figure 2: performance of one of the computation phases in the current design. The different lines represent the time taken for two files of diferent sizes

3 MPI-IO and the vector topological data input module

3.1 Initial data input

The first problem to be solved is how to use MPI-IO to read in the initial data during the Sort phase [2] of the vector-topological data input module. As was mentioned in section 2.1.3, in order to achieve high performance results, we need to specify as much as possible, the I/O request patterns of the involved processes. This presents some difficulties that can be better understood by a brief explanation on the representation used to store the initial data.

3.1.1 Input Data Format

The initial data is represented by a set of records stored in NTF files[3] . These files are in ascii and to a certain degree, are human readable. In order to specify an access pattern there has to be information on how the data is structured so that it can be divided between the cooperating processes. In NTF files, some records are of variable length, and there is no specified position for each type of record, ie. each record can be stored anywhere within the data section of the file. One way to distribute the data between the Pool processes is to split the data into equal size chunks. However, record splitting must be avoided, as they should be read in their entirety into a Pool processes' buffer.

3.1.2 NTF mapping

One way to specify the Pool proccesses access patterns is to create a map of the NTF file containing information of the file structure, and use it to specify the requested access patterns. Two types of mapping were implemented, contiguous and non-contiguous. The first type maps all the records in the NTF file, storing only their starting positions. Each record's size can be easily computed from this information. The non-contiguous mapping, stores only the information related to the relevant records used by the Pool processes. The reason behind the creation of the second method is to see if the I/O can be improved by reading as little data as possible. Two tools were created, one for each type of mapping, both taking around 30 seconds (on a Sun Enterprise Server 3000) to generate a map file from an original NTF file of 19 Mb, used during the performance tests. The resulting map files have sizes of 8% of the original test file for the contiguous mapping, and 12% for the non-contiguous one. The usage of these tools is described bellow:

1. Contiguous - mapntfc [-b <input buffer size>] *file.ntf file.cmap*
2. Non-contiguous - mapntfnc [-b <input buffer size>] *file.ntf file.ncmap*

Both tools will generate a map (*file.cmap* or *file.ncmap*) from the original data (*file.ntf*). The <input buffer size> option is given in Kb and specifies the amount of memory that should be allocated for the input buffer. It hasn't been tested whether large buffers improve the performance of these tools.

A pseudocode of the implementation of these tools is presented in Appendix A.

3.1.3 Design

Armed with the mapping information the Source process can then assign blocks of data to the various Pool processes, ensuring that each record is processed only once. It also allows the specification of the access patterns required to use MPI-IO efficiently.

Here's a general description of the data distributing scheme:

1. The Source process reads the map file, assesses the amount of data to be processed, the number of Pool processes and the maximum block size for each process.

2. With the above information, a set of blocks is computed to match the number of existing Pool processes and the maximum block size (ie, in each block assignment computation the number of blocks is equal to the number of Pool processes and the block size is as close as possible to the maximum block size each process can take).

3. A block is assigned to a Pool process only when a corresponding "ready" signal is received by the Source process (ie. each Pool process explicitly states it's readiness to process more data). Each block is then read directly from disk without any intervention from the Source.

4. After all the blocks from the set have been assigned, the Source process goes back to point 2 until no more data is left to process. Meanwhile the Pool processes access the NTF file through MPI-IO with enough information to specify their access pattern using collective IO function calls. After finishing the requested processing they issue a "ready" signal to the Source process.

5. The Pool processes terminate after receiving an appropriate signal from the Source process indicating no more data to process.

For a more detailed description please refer to Appendices B and C.

The next figure illustrates the proposed design, where the data is read directly by the Pool processes improving communications overhead.

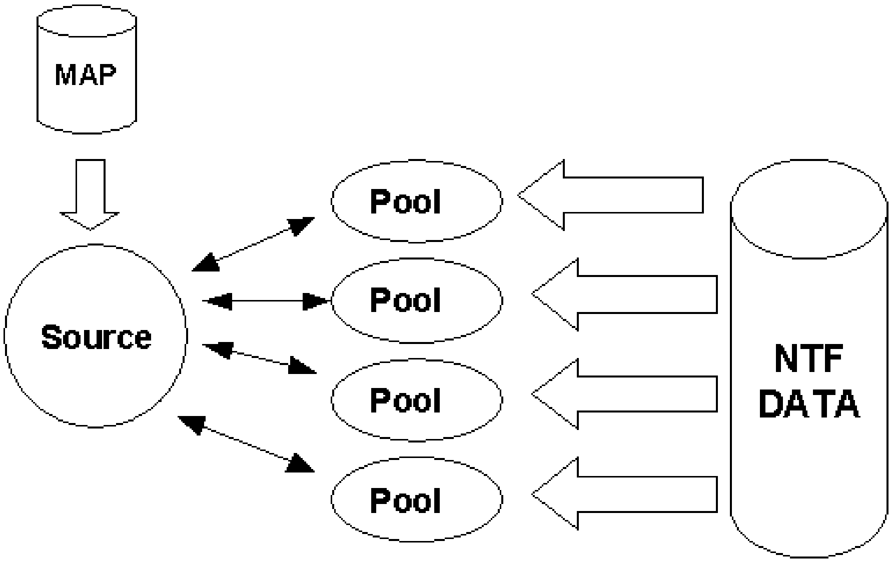

Figure 3: information flow in the proposed design

4 Results

In this section, some results are presented in relation to performance of the proposed design. All tests where carried out on a Sun Enterprise Server 3000 which is a multi-user system therefore these results are subject to variations according to the machine load.

4.1 Comparison of Unix standard I/O calls to MPI-IO

The following table shows the time taken (in seconds) by 16 Pool processes to read a 19 Mb NTF file using both mapping methods (contiguous and non-contiguous) with Unix I/O calls (classified as *Level 0*) and MPI-IO collective calls (classified as *Level 1*). Several block sizes are used to illustrate their importance in I/O performance.

Block size	C UNIX	NC UNIX	C MPI	NC MPI
64Kb	12.91	12.53	191.41	185.42
128Kb	8.83	8.71	104.06	95.26
256Kb	5.15	4.71	55.83	54.43

Some interpretations follow:

1. The cost of communications involved in using the MPI-IO library seems to outweigh all the benefits of generating less file requests to the system.

2. The usage of large block sizes alone improves the I/O performance without using MPI-IO. This suggests that the proposed design might be useful to read in the NTF data without resorting to MPI-IO, if large enough block sizes are used. However very large block sizes might affect the load balancing on the Pool processes, as the data allocation approaches a *Block* distribution instead of the more desired *Cyclic* one.

3. The non-contiguous version seems to perform slightly better, as some of the irrelevant records are skipped during the block allocation, reducing the total amount of data to be read by the Pool processes.

The fact that MPI-IO performs so badly is not so suprising as only a *Level 1* type of access was used. If very small block sizes are used (eg. 8 kilobytes) then a *Level 3* type of access must be used, meaning a cyclic allocation has to be computed by the Source process, thus reducing potential load balancing problems that might be associated to larger blocks. With this type of distribution MPI-IO should perform better than standard Unix type calls[4].

4.2 Suitability of the proposed design using standard Unix I/O

This section looks into the possibility of using the proposed design with standard Unix calls, ie. the assigned blocks are read by the pool processes using functions such as *fseek()* and *fread()*.

4.2.1 Testing method

As the design only deals with reading the initial data, some liberties were taken to perform the required tests:

1. The number of processors is simulated by creating as many processes as needed.
2. In order to simulate the amount of work done by each Pool process, a call to a *sleep()* function is performed using a time argument depending on the size of each read block.
3. The computation time was fixed as 1 second per 64Kb of the used test file. At this rate the total amount of time taken by one process would be around 297 seconds.

These *"assumptions"* permit the running of tests with more Pool processes than actual processors, because no actual work is performed besides reading the data.

Several block sizes are used to test the potential scalability of the proposed design. During these tests no effect on load balancing was taken into account, suggesting a repetition of these tests using actual operations carried out on the data.

Figures 4 through 6 show the degradation of the potential parallel efficiency for different block sizes.

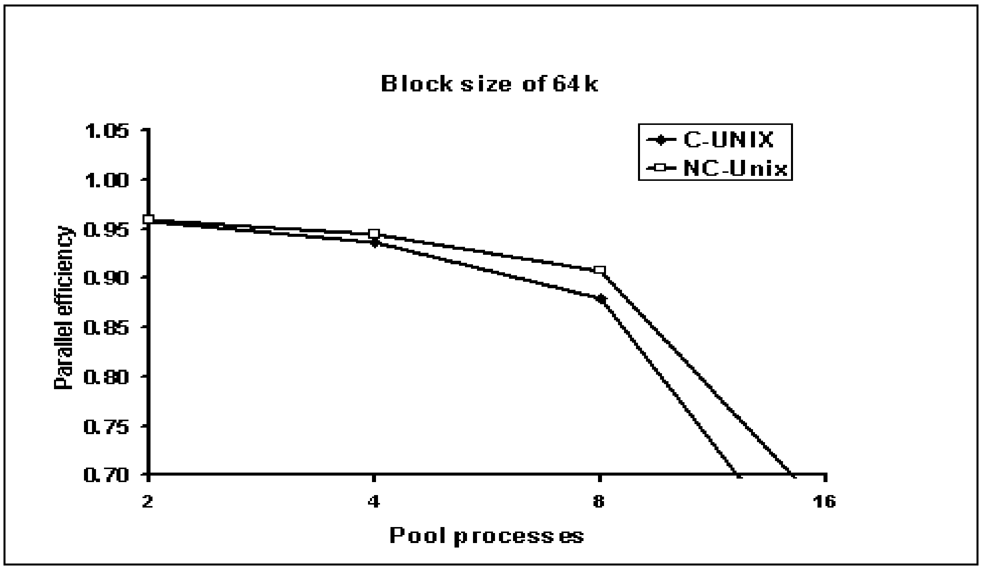

Figure 4: Parallel efficiency with a block of 64Kb

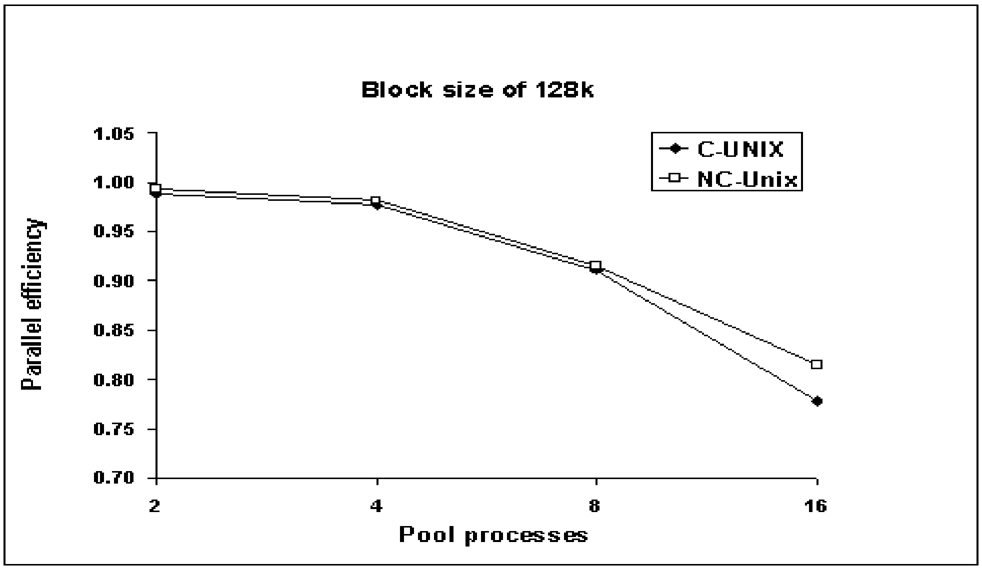

Figure 5: Parallel efficiency with a block of 128Kb

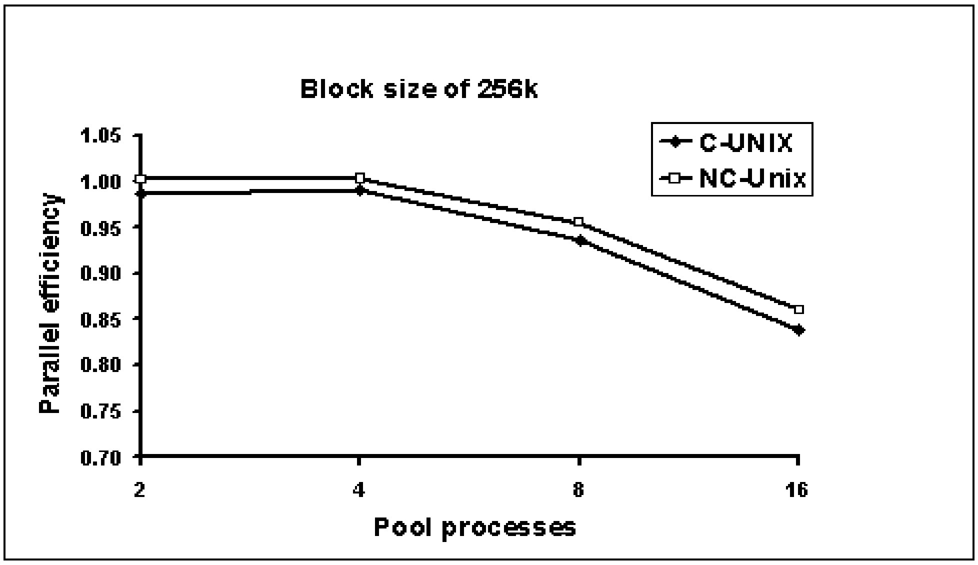

Figure 6: Parallel efficiency with a block of 256Kb

4.2.2 Discussion of results

It is evident from these figures that the scalability increases accordingly to the block size used. As was stated before, these results might be incorrect if load balancing is affected with the increase of block size, however as the results indicate, a good scalability can be obtained by using block sizes as small as 128Kb (less than 1% of the used NTF file) which might be small enough to achieve an adequate load balancing. These results present strong evidence that this design might be useful to improve significantly the input of the initial data even though MPI-IO is not used.

5 Conclusion

5.1 MPI-IO and parallel IO

MPI-IO seems to solve problems related to parallel I/O of heavily interleaved information, ie when a fine grain cyclic distribution of data is used. To take advantage of this library, the access patterns of the intervening processes must be known and specified to the I/O function calls. This implies a detailed knowledge of the structure of the data accessed by each process.

5.2 Finding structure in NTF files

As NTF files don't have a regular structure, a map has to be created containing details on the position size[1] and type[2] of each record stored in a given file. This information can then be used to generate the Pool processes' access patterns.

5.3 MPI-IO performance in the proposed design

It was found that the use of MPI-IO with a *level 1* type of access doesn't offer advantages over standard I/O calls. This can be explained by the overhead incurred in the communications phase of the collective MPI-IO calls used. In fact the usage of simple UNIX file functions to read the assigned blocks generates a small number of large contiguous requests to the file system achieving the type of optimisations that MPI-IO attempts.

5.4 Suitability of proposed design using standard file functions

If load balancing is not affected by block sizes of up to 64 or 128 kilobytes then this design might be useful to improve I/O performance, using standard file functions to read the data. However if a cyclic allocation consisting of small data chunks is needed to guarantee an adequate load balancing, then the allocation method has to be changed and a *level 3* type of access must be used in conjuction with MPI-IO.

[1] Only used in non-contiguous allocation

[2] Only used in non-contiguous allocation and useful for the proposed Pool computation optimisations only

6 Future work

Due to time constraints, a large number of tasks where left undone. These include:

1. Determining the relations between load balancing and the size of the blocks assigned to pool processes. A weak correlation of the two factors means that very large blocks can be used with the current design, if however, this is not the case, then a different block assignment algorithm should be tried to perform a cyclic distribution of the data, in which case MPI-IO must be used.
2. Investigating the possibility of using the mapping data to optimise the operations carried out by the Pool processes. This could be achieved by sending the mapping information of the corresponding assigned blocks to the pool processes, which could use this to more efficiently find and process the records contained in the read blocks.
3. Improving the remaining I/O sections of the vector-topological data input module, namely the output and gathering of results from Pool processes. A proposal to improve output performance follows.

6.1 Design proposal to improve output to global files

The Source process could also be used to coordinate the output of data to global files, using these protocols:

1. Pool processes issue a request to the Source process stating the need to write to an output file. The amount of data to be written is specified in these requests.
2. The source process keeps track of the writing position within the file. This position is transmitted back in response to the access requests received and updated using the size of the same requests, thus ensuring that Pool processes don't write in overlapping positions.

If large enough block sizes are used to output the results then writing performance should be as good as the reading performance presented above. Also, if each output block is partitioned into smaller units (much like the records in NTF data), a map can be generated by each Pool process as the output data is produced, and then saved to a global file using the same mechanism, or transmitted to the Source process that will save it in the appropriate file.

7 Appendices

7.1 Appendix A - Pseudocode for the maping tools

7.1.1 Non-contiguous version - mapntfnc

```
Open file.
Read data into buffer
Log data size on map file
Find first record type by taking the first two
   characters in the buffer
While not end of data
   Find record position using the current
      position in the buffer
   Find the record's size by searching for
      the next record's beginning
   Log record type, size and position on map file
   Find next record's type using the next two
      characters in the buffer
End while
```

7.1.2 Contiguous version - mapntfc

```
Open file.
Read data into buffer
Log data size on map file
Skip first record type indicator in the buffer
While not end of data
   Find record position using the current
     position in the buffer
   Use size of the current record to
     update the buffer position
   Log record position on map file
End while
```

7.2 Appendix B - Pseudocode for the Source process

7.2.1 Main section

```
Set number of blocks to number of pool processors
Create array of blocks with the above quantity
Read the map file
While not end of map data
   Compute new block size
   Compute a partition and store it in the block array
   Distribute blocks to the pool processes that
      have sent a _ready_ message
End while
Send _death_ messages to the Pool processes
```

7.2.2 Partition computation

```
Clear the received block array
While not end of block array and not end of map data
   Block starting position = last block ending position
   While the end of the block is not reached
      Extract next record and include it in block
      Add record size to total size
   Block ending position = block starting position + total size
   Store block in array
   Step to next block
```

7.3 Apendix C - Pseudocode for Pool processes

7.3.1 Unix file calls

```
Allocate a buffer to hold the assigned blocks
Open NTF file
While _death_ message not received
   Send a _ready_ message to the Source process
   Receive an assigned block
   If the block is not a _death_ message
      Find the block size
      Seek the file for the beggining of the block
      Read the block
      Process the read data
End while
Close NTF file
```

7.3.2 MPI file calls

```
Allocate a buffer to hold the assigned blocks
Open NTF file collectivelly
While _death_ message not received
   Send a _ready_ message to the Source process
   Receive an assigned block
   If the block is a _sync_
      Issue a collective read of 0 bytes to ensure that all
         processes make the collective call
   if the block is valid
      Find the block size
      Read the block collectivelly at it’s position
      Process the read data
End while
close NTF file
```

References

[1] Michal Rewienski:Parallel vector-topological data input for Geographical information systems, SSP Report (1996)

[2] Terence M Sloan,Steve Dowers, Timothy J Harding, Evan Welsh: Chapter 8 - Parallel Vector Data input, EPCC-PAP-GIS-CH-8 0.2 (1995)

[3] Timothy J Harding, Michael J Mineter, Terence M Sloan: GIS Operations Interface Functional Specification, EPCC-PAP-GIS-INTERFACE-FS 1.2 (1993)

[4] Rajeev Thakur, William Gropp, Ewing Lusk:A Case for Using MPI's Derived Datatypes to Improve I/O performance, Argonne National Laboratory (1998)

Eurico Moita
undergraduate student,
Department of Informatics,
Faculty of Sciences of the University of Lisbon, Portugal
E-mail: i21324@caravela.di.fc.ul.pt

SSP Supervisor: Terence M.Sloan,
Edinburgh Parallel Computing Centre