

Parallel Elcirc 3.11 MPI Implementation

Parallel Elcirc Team

Department of Computer Science, University of Tromsø

Department of Computer Science and Engineering,

OGI School of Science and Engineering, Oregon Health & Science University

1 Introduction

This report describes the parallel MPI implementation of Elcirc version 3.11. It is based on a parallel version using PATHS, written by John Markus Bjørndalen. Please refer [1] for the design choices made when parallelizing Elcirc.

Figure 1 gives an overview of the architecture of the sequential, PATHS, and MPI version of Elcirc. The Elcirc application code is unchanged for all versions, with the exception of a few calls to the communication library. The communication API is the same for the PATHS and MPI version. Code to extract data from arrays, and to do necessary manipulation before communicating is reused from the PATHS implementation. Also, functionality for configuring who communicates with who, and what data to communicate is reimplemented in the MPI version. In the MPI version, MPI is used to send and receive data.

The rest of this report proceeds as follows. Section 2 gives a short summary of the parallel PATHS version. The MPI implementation is described in section 3. Experiments are described in section 4. Section 5 concludes and outlines future work.

2 PATHS Implementation

This is a summary of the Elcirc version described in [1].

Before Elcirc was partitioned the sequential code was modularized and cleaned up. This simplified the parallelization, and also the port from PATHS to MPI.

Elcirc was parallelized using domain decomposition. All Elcirc stages are parallelized. The data set is partitioned into smaller pieces containing a partition and ghost regions. The parallelization required few changes to the source code.

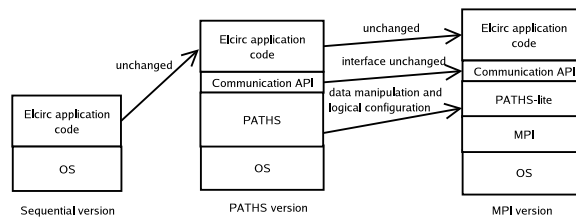


Figure 1: Architecture of different Elcirc implementations.

The ghost regions are exchanged by processes responsible for neighboring regions in the grid. Important for scalability of the application is the size of the ghost-regions relative to the size of the region. A larger ghost region means that more data must be communicated.

The parallel and sequential version use the same source code. Annotations are added to the source code, which are used by a pre-compiler to add communication functions to the code. These functions read updated data from neighbor regions and apply them to the ghost regions, and others extract data used by neighbors and send it to these neighbors.

The PATHS/PastSet systems is used for communication. PastSet is a structured-shared memory system, while PATHS allows adding wrapper code to, for example, manipulate the data written to the structured-shared memory.

2.1 Source Files

In this section we give a short code walk-through of the files used in the PATHS versions of Elcirc. Some function names are changed from Bjørndalens implementation to provide more generic naming scheme for the communication API.

The Elcirc code is divided into several files. Each file has code for one module (such as backtracking). The code relevant for the parallelization can be found in the main loop in `main_loop.f`. The communication is implemented by the functions `comm_layer_read_vars` and `comm_layer_write_vars`. Also the function `comm_layer_exchange_vars` can be used to combine several `comm_layer_read_vars` and `comm_layer_write_vars` calls.

In addition there are function for tracing (`ts_tamp`), initialization (`comm_layer_initialize`), and finalization (`comm_layer_finalize`).

!@P is a preprocessor directive. After running the preprocessor (`transcripts/mpi-preproc.py`), a new Fortran file, `pstst.f`, will be created in `bin/`. In this file the calls to `exchange_vars` are replaced with calls to `comm_layer_write_var` and `comm_layer_read_var`. The implementation of the `comm_layer` functions can be found in `extralib/paths_support.c`.

`Write_var` does a write, using the PastSet operation `ps_move`, to a PastSet element. While `read_var` does a read from a PastSet element. These functions allows to send and receive data in different arrays. The PATHS approach allows the data written (and read) to be processed by wrapper code before writing to (or reading from) the PastSet element. In Elcirc, two wrappers are used, `subsample` and `fork`.

`Subsample` and `fork` implements an API, and allows us to configure the mapping of the operations and communication links to the clusters in use. `Subsample` extract a subset of entries from the array being sent, and applies updates to the received array. `Fork` is used to optimize and simplify the usage of `subsample`. The code for these wrappers can be found in `paths/c/pastset-xfun-subsample.c` and `paths/c/pastset-xfun-fork.c`.

2.2 Tools

Elcirc-PATHS provides several tools for partitioning the grid, computing ghost regions, finding the different matrix-indexes of the ghost-region elements, and setting up the communication paths used by PATHS.

To create a configuration for N processes the following tools are run:

1. *readtst* reads the grid from the Elcirc input files and converts it to an intermediate format.
2. The intermediate files are used by *Metis*¹ to divide the grid into N parts.
3. The Metis output files are used by the *find_overlap_new.py* tool to create the ghost regions. This tool also writes information about the ghost regions to files used by the PATHS subsample wrapper during initialization.

Each process has a directory where the resulting configuration files are stored (run/partitions/p0[1-N]).

3 Elcirc-MPI

In addition to rewriting the code to use MPI message passing instead of the PastSet structured shared memory, porting to MPI consisted of writing a high-level library that use MPI for communication. This consisted of moving part of the PATHS functionality to a library that is called before and after the MPI communication operations

3.1 Elcirc Source File Changes

Paths_support.c is replaced by comm_layer.c. The code for the PATHS subsample wrapper has been copied to comm_functions.c. The fork wrapper functionality is implemented in comm_layer.c. In addition some debugging functions in utils.h are used.

To the main function in main.f, three lines of code are added: (i) the mpi header file is included, (ii) MPI is initialized by calling MPI_Init, and (iii) there is a call to comm_layer_chdir.

Comm_layer_chdir changes the working directory of the current process to the directory where its input files are. This could have been implemented in Fortran (if I knew how to do it). Also, MPI should be initialized in comm_layer_init (if someone knows how to send argc and argv from Fortran to C).

3.2 Tools

Two PATHS tools are modified. The first, mpi-preproc.py, is the PATHS preprocessor with minor changes in exchange_vars. Also, the output filename is changed to mpitst.f.

The second, create_pathmap.py extracts and creates information used to set up the send and receive paths. It is based on the PATHS find_overlap_new.py tool. In addition to changing the output format, three changes are made due to the different programming models used: (i) instead of writing data to a named tuple space, data is sent to a process with a given rank, (ii) there is no information about which hosts processes are located on, and (iii) the only operations that can be done on the sent/received data are extract from matrix and negate.

Create_pathmap creates, for each process, one file for send paths, and one for receive paths. Both files have similar syntax. The first line gives the number of matrices (M), and the following M lines gives for each matrix the name and number of paths. For each path we store information about: the matrix name, destination or source rank, the tag, the MPI type, the name of the ghost region index file, and the name of the reverse index file.

¹Available at: <http://www-users.cs.umn.edu/~karypis/metis/>

The index files contain information about which matrix elements are part of the ghost region and which elements need to be negated.

3.3 Communication Layer

The Elcirc code, written in Fortran, does not require any knowledge of the communication layer. The programmer only see calls to distribute and receive changes to a matrix (`write_var` and `read_var`). The communication layer can be changed without modifying the Elcirc source code. Also, different configurations can be tested without modifying the communication layer or application code (e.g. to experiment with ghost region sizes).

For all matrices containing data to be exchanged, a preprocessor adds calls to the `comm_layer_read_vars` and `comm_layer_write_vars` using the matrix name as the only argument.

The matrix name is used by `write_var` to find all processes that should receive a ghost region from the given matrix. To each receiver, we have a *send path*, that is identified by the MPI rank of the receiver. Before sending the data, ghost elements are extracted from the matrix and copied to a buffer. For some matrices the ghost element data is also negated. There is one buffer for each path, that is used in the MPI send and receive functions.

`Read_var` is similar to `write_var`. It does a lookup for the *receive paths* for the given matrix, receives data to a buffer, negates the data (if required), and copies the data to the ghost elements entries in the matrix.

The extract and negate functions are required since the ghost region elements are scattered unevenly in a matrix making it difficult to use e.g. derived types. The functions to extract and negate the data are from the PATHS subsample wrapper.

The path lookup, and data manipulation is a simplified implementation of PATHS (without all the flexibility offered by the PATHS system). It allows to experiment with different communication configurations, ghost region sizes, and partitioning models.

The send and receive paths are implemented using a list. Each node in the list is identified by the matrix name, and it has a list of all send/receive paths for the given matrix. A send (and receive) path is identified by the receiver (or sender) rank. For each path there is an array with the ghost region indexes, an array with negate indexes, the MPI tag and MPI type, pointers to the extract and negate functions for the given type, and a buffer for storing messages to send and receive.

The lists are initialized by reading information from files created as described in the next section.

The send and receive paths are divided into two lists for ease of implementation. Provided that there always is an exchange of ghost regions between two processes these could be combined.

In the initial implementation `MPI_Send` and `MPI_Recv` were used. However, using (possibly) blocking sends with large messages caused the application to deadlock. For large messages the LAM/MPI implementation does not send all data before the receiver has called the receive function. Thus all processors are blocked in the send call. This could probably be avoided if the unexpected message buffer size was larger.

There are many possible solutions to the deadlock problem: (a) `MPI_Sendrecv` could be used, (b) using non-blocking send (`MPI_Isend`) and/or receives (`MPI_Irecv`), (c) do a non-blocking receive (`MPI_Irecv`) before each send and waiting for data after

Process	Elevation	Salinity	vertw.63	fort.64
p01	0.021155	1.705350	0.011468	0.093425
p02	0.043924	5.161460	0.008609	0.289949
p03	0.016745	2.669370	0.012025	0.049961
p04	0.006958	0.000000	0.000367	0.005617
p05	0.009383	0.000000	0.000825	0.016824
p06	0.002868	0.000000	0.000193	0.003078
p07	0.012840	0.000000	0.001939	0.015405
p08	0.014536	0.000000	0.001140	0.027396

Table 1: Maximum differences for each process.

the send (MPI_Wait), or (d) reordering the sends and received.²

MPI_Sendrecv could be used, by combing read_var and write_var. Similarly, doing a MPI_Irecv before a MPI_Send and MPI_Wait after the send, could also be implemented by combining read_var with write_var. Reordering the sends and receives would require to rewrite write_var and send_var to not send data to all recipients for a given matrix. Instead the sends and receives have to be reordered such that deadlock cannot occur. This could probably be done by the preprocessor. However, all would require a strict ordering on who communicate with who, something which becomes complicated since we have graph of communicating processes (and not only a tree).

The chosen solution is to combine non-blocking sends with blocking receives. Write_var uses MPI_Isend to send messages. After the send, the pending flag for the send path is set. Read_var uses blocking receives (MPI_Recv). Between two calls to write_var with the same matrix as argument, comm_layer_commit has to be called. This functions waits until all pending sends have completed (using MPI_Wait).

Using non-blocking receives caused the program to crash. I do not know why.

Each matrix has a unique (integer) identifier that is used as tag for all messages containing data from that matrix. This is not strictly necessary, but has been implemented for more flexibility in which order messages are sent and received.

4 Experiments

The implementation has been tested with different number of processors. Elcirc runs until completion and output files are created.

The performance of the different communication structures has not been measured.

4.1 Accuracy

The PATHS version and MPI version were run on one 8-way host. There was one process per CPU, and for the inter-process communication PATHS or MPI were used. The same input files, ghost regions and partitions were used.

The resulting output files were compared using the compare-results tool. The maximum differences for each process are given in table 1, while table 2 gives the average differences.

²Based on a discussion in the LAM/MPI general users mailing list <http://www.lam-mpi.org/MailArchives/lam/msg00608.php>

Process	Elevation	Salinity	vertw.63	fort.64
p01	0.000536	0.015981	0.000007	0.000197
p02	0.009080	0.214192	0.000046	0.002100
p03	0.002991	0.026335	0.000018	0.000739
p04	0.000131	0.000000	0.000001	0.000038
p05	0.000912	0.000000	0.000004	0.000266
p06	0.000161	0.000000	0.000000	0.000046
p07	0.002169	0.000000	0.000004	0.000254
p08	0.002078	0.000000	0.000007	0.000442

Table 2: Average differences for each process.

5 Conclusion

This report has described how the PATHS communication system used by a parallel version of Elcirc 3.11 has been replaced by MPI. Some of the PATHS functionality and code has been reused in the MPI implementation.

Due to the modularity of the application the communication layer API could easily be implemented using MPI. However, the communication structure had to be reorganized since LAM/MPI deadlocked when large messages was used.

Overall, the work involved porting from PATHS to MPI was small (about 3 days). We believe parallel Elcirc 3.11 can be used as a reference-model for the parallel Elcirc 5. Also, many of the tools developed for Elcirc 3.11 can be used with some modifications.

References

- [1] BJØRNDALLEN, J. M. *Improving the Speedup of Parallel and Distributed Applications on Clusters and Multi-Clusters*. PhD thesis, Tromsø University, 2003.