MPI Visualisation

Thomas RICHARD

Academic year 2010/2011

MSc in High Performance Computing
Edinburgh Parallel Computing Centre
The University of Edinburgh
Year of Presentation: 2011
Abstract

The MPI Visualisation project aimed to generate a real-time visualisation tool for MPI codes. Standard tools, as Vampir or Scalasca, use post mortem analysis, they generate files with information that they analyse later. This project proposes another way of gathering information about an MPI program. It uses the MPI Profiling interface to wrap the MPI functions, collect the data and display them in a separated display at the same time that the program is executed.

The original design was adapted in order to decouple the Profiler, that is gathering the information, and the Interface, that displays it, in order to increase modularity. The resulting tool aims to be easy to use, to be accessible to MPI learners, and successfully manage to handle most of the MPI point to point communications functions. It also proposes a memory registration system. That will allow a code to register arrays and will then display what parts were accessed during communications. Some derived MPI datatypes are also supported, such as Vectors and Contiguous, in order to be useful for the first steps of programming with MPI.

Finally the tool was tested on four different codes, developed as Message Passing Interface courseworks earlier in the year. The tool proved to be adaptable and robust, as three of the codes were not written by the author. The codes were using derived datatypes, multiple communicators and several arrays (to the order of 100,000 elements per processors) on a shared memory cluster of 16 processors. As it was developed as a prototype, the project still has missing features and a few drawbacks, among them heavy memory consumption for the Interface. Nonetheless it could be used to visualise simple code and could help learner understand their first errors with MPI.
## Contents

**Introduction**

1 The MPI standard and the project motivations .......................... 2
   1.1 What is MPI used for? ........................................... 2
   1.2 Existing approaches, motivations and goals .......................... 4
      1.2.1 Project motivations ........................................ 4
      1.2.2 Visualisation tools for MPI ............................. 5
      1.2.3 Project goals .............................................. 7

2 Profiling an MPI program .................................. 9
   2.1 The MPI Profiling Interface .................................. 9
   2.2 Profiler organisation ..................................... 10
      2.2.1 Communications ........................................ 11
      2.2.2 A template linked list in C ............................. 18
      2.2.3 Registering MPI communicators ......................... 20
      2.2.4 Registering memory ................................... 21
   2.3 Limitations ................................................. 23

3 The Graphical User Interface .................................. 24
   3.1 The Qt framework ............................................ 24
      3.1.1 Qt root: theQObject class .................................. 25
      3.1.2 User Interface and Graphics in Qt .......................... 27
      3.1.3 Threading environment ................................... 27
      3.1.4 Other tools of the Qt SDK ................................ 28
   3.2 The Interface organisation .................................. 30
      3.2.1 The GUI organisation ................................... 30
      3.2.2 Connection to the Profiler ................................ 32
      3.2.3 Communications with the Profiler .......................... 34
      3.2.4 Statistics ................................................. 38
      3.2.5 Communications visualisation ............................... 41
      3.2.6 Memory visualisation .................................... 43
      3.2.7 Export information ...................................... 45
   3.3 Limitations ................................................... 46

4 Achievements and Future work ................................ 48
4.1 Profiling usages ........................................ 48
   4.1.1 Technical usage .................................. 48
4.2 How to use the tool ..................................... 51
   4.2.1 Code modification ................................. 52
   4.2.2 Running the Interface ............................... 53
4.3 Applications results ................................... 55
   4.3.1 Simple codes .................................... 56
   4.3.2 The MPP coursework ............................... 57
4.4 Requirements compliance ................................ 61

Conclusion ................................................. 63

Bibliography ..................................................... 65

A Original Project Proposal .................................. 66
B Project Functional Requirements ............................ 68
C Code Listings .................................................. 69
D The test codes ................................................ 75
E Memory consumption ......................................... 78
F Messages in strings or messages in a datagram ............ 80
G Functionalities enhancement: examples .................... 83
   G.1 Adding support for MPI_Send ....................... 83
   G.2 Adding support for new MPI Datatypes ............ 88
List of Tables

3.1 Different port formatting for MPICH2 and OpenMPI .......................... 33
3.2 The different attribute usage of the MessageInfo according to the message. 37
4.1 Array size for each process and their overall total and halo size. .......... 57
E.1 Executable size and Memory (in Bytes) usage for the hello world code. 78
E.2 Executable size and Memory (in Bytes) usage for the mulcom code. .... 79
E.3 Executable size and Memory (in Bytes) usage for the MPP casestudy code. ................................................................. 79
F.1 Comparison of the size of message in a string or datagram. ............... 82
List of Figures

1.1 Several tools interfaces. ........................................... 6

2.1 Evolution of the processes organisation ................................ 12
2.2 Vector datatype information (memory and matrix view). ........... 15
2.3 The registrations and initialisation messages. ..................... 16
2.4 The communication messages. .................................... 17
2.5 Communications between Profiler and Interface for a call to MPI_Ssend(). 18
2.6 Double Linked List and its Template Cells, representation. ...... 19
2.7 The Communicator and Request information data-structures. ...... 21
2.8 The datatype information data-structure. .......................... 22

3.1 The Qt modules ....................................................... 25
3.2 Sequence diagram of calls for a thread implemented with QThread. 28
3.3 The Interface classes organisation. ................................ 31
3.4 The Starter dialogue: connection to the Profiler .................. 33
3.5 MPIWatch communications with the Profiler. ...................... 35
3.6 The MessageInfo class diagram. .................................. 37
3.7 The several MPIWatch signals propagation. ....................... 39
3.8 The three statistics information windows. ........................ 40
3.9 An occurring MPI_Ssend() synchronised with the user. .......... 42
3.10 A vector datatype memory and matrix view. ...................... 44
3.11 The Datatype classes tree. ...................................... 44
3.12 The XML information tree. ...................................... 45
3.13 The MVC organisation design. .................................... 46

4.1 The correct and wrong halo swapping for the traffic model. ....... 49
4.2 Matrix transposition visualisation. ................................ 50
4.3 The message in a ring communications view (8 processes). ...... 54
4.4 The mpidisplay memory consumption for different traffic model’s size. 58
4.5 The mpidisplay memory usage for different implementation of the MPP coursework. ........................................ 58
4.6 The deep halo swapping steps. .................................... 60

D.1 The ring communication pattern with 4 processors. ................. 75
D.2 The Traffic model on 4 processors: data exchange. ............... 76
D.3 Hallo swapping on a 2D decomposition on 4 processes. .......... 77
Acknowledgements

Thanks to David Henty for his advices and support during the whole project.

Special thanks to Lesleis Nagy who helped me not getting mad while debugging and without whom this dissertation may have been less English.
And to Paul Ross for his coffee drinking advices and support.
Introduction

Science always needs more computing power. Solving a problem is not always good enough, solving it fast is often what is required. High Performance Computing (HPC) is at the boundary of Science and advanced computation techniques, using large purposely built systems. Such systems are massively parallel, with thousands of processing units on a single machine, achieving petascale\textsuperscript{1} performance. The goal for 2018 is reaching exascale - 1000 times more than petascale - this is only achievable with efficient inter-processor communication.

The state of the art technology uses accelerators and multi-threading on a node and communication between those nodes on very fast networks. Even if the interpretation of a node is somewhat unclear (it is defined as a processing unit, that can either be a single core, or the whole processor) both techniques will always require the use of reliable and fast message passing technology between nodes. The Message Passing Interface (MPI) standard is a necessary tool for the computations of tomorrow.

Being able to write an efficient program running on several thousands of nodes requires good knowledge, built on learning and experience. Learning is the key step where one familiarises oneself with the main concepts; experience is the careful exploration of the advanced possibilities. Learning MPI correctly is needed to perform efficient massive processing.

The goal of the MPI Visualisation project is to help the learner to visualise the underlying actions which occur when using MPI libraries. It is important to provide learners a tool that can help them find errors and understand them. Complete and powerful tools exist on the market to visualise and analyse MPI codes; but none are simple to use, and therefore easy to introduce in the learning process. Moreover they are based on the analysis of a complete run of a program and will not help understand why a program did not complete (if it crashes or deadlocks) - this is often the problem for a beginner.

This project aims to solve some of the problem learners can have by providing a real time visualisation of a code using MPI. It has to be easy to use, simple and reliable. This document proposes to introduce the MPI standard and the motivation for the project in more detail. Then the core parts of the project will be discussed, one that analyses a code and one that displays the information. Finally the achievements will be developed, followed by a conclusion to the project and proposals for future work.

\textsuperscript{1}petascale: $10^{15}$ floating points operations per seconds (flop/s)
Chapter 1

The MPI standard and the project motivations

1.1 What is MPI used for?

The natural way to improve the computing power of an installation has been, for a long time now, to simply duplicate the hardware as many times as possible. The Shared Memory idea is to share a global memory between multiple computing units, while the Message Passing principle is to communicate between them.

However as Shared Memory systems are limited to few processors (on the order of hundreds of processors), the Message Passing philosophy allows, in principle, the use of an enormous number of them (a hundred thousand cores are used on current best machines of the world [1, June 2011 Top 500 list]).

At the beginning of the 1990s several Message Passing libraries were in existence on the supercomputer “market”. Therefore the development of code was tightly linked to the hardware used - as vendors usually developed their own libraries - and the portability of ones’ code was close to zero. Instead of taking an existing systems as standard, a group of individuals from major vendors and universities formed the MPI Forum. They started to compile the MPI standard in 1992 and the final draft was presented in November 1993; it was later revised as the version 2.2 in September 2009 [2, Background of MPI-1.0].

The standard defines the major goals to reach for an MPI implementation. To do so it contains a list of functionalities, with detailed actions and use cases. The implementation is left to the vendors, as the goal of the MPI Forum is to design an efficient Message Passing library that is usable on any system for several development languages (Fortran, C and C++ are the principle ones).

Today the two major implementations on the market are MPICH2 and OpenMPI. Both
are open source and widely used; they represent the state of the art of portable and optimised Message Passing libraries. They are compliant to the MPI 2.2 standard and provide multithreading enhancement, along with many other advanced features.

The type of code using MPI

MPI is a comprehensive standard, developed to use big computing structures, usually hosted by national-wide research laboratories or international companies (HPC systems maintenance cost is high). It is therefore used in many fields, with different types of code. However it could also be used in smaller clusters, like the one provided by the Edinburgh Parallel Computing Centre (EPCC) for the MSc in HPC (called Ness).

This vast portability make the type of code that uses MPI very different. The UK’s national super computer, HECToR, is mostly oriented toward Scientific code. The provided example list mostly includes Chemistry and Physics research (although some advanced modelling on dinosaurs or even how Elephant Man used to walk have been implemented) [3]. MPI is used in most - and certainly all of large scale - of the mentioned project. As an example MPI is explicitly used in SPRINT, the Forecasting Faster or the New tricks from old oil (the latter uses the CASTEP library) [4, 5, 6, 7]. However not only final application uses MPI, several libraries exist and are used with MPI. It is possible to mention FFTW or PETSc as certainly two of the most used libraries built on MPI in Computational Science [8, 9].

Learning MPI

MPI is a standard software package. Using it is fairly easy from the library perspective. The complexity in learning MPI is related more to the concepts it provides. Even if it is distributed as a library, the implementation usually provides compilers (mpicc for C code, mpif90 for Fortran and mpiCC for C++) that are simple scripts that add headers inclusion and linking options to the system’s compilers.

On the other hand the concepts provided by MPI are important to understand. As expected most of the key concepts are related to communications. There are three types of communication to understand to have a basic grasp of MPI:

- Synchronous communication, that cannot complete before both sides perform the action (i.e. sending and receiving is done at the same time for the two processes).
- Non-blocking communication, meaning that at least one of the actors is not waiting for the action to complete to continue. This generates requests that have to be waited on in order to ensure that data was transferred, more importantly to allow reuse of the data storage used for the communication.
- Global communication that requires all the processes to perform the communication in order to complete.

The two other features prosed by MPI that are interesting for a learner are the creation of derived datatypes and MPI-IO. The former allows the user to define access patterns
on blocks of memory, like vectors (access a certain number of element regularly) or subarrays (access a sub-part of an array). The latter is used for efficient parallel Input/Output implementation, that allow the user to read and write in files from several processes.

It is also very important to understand how a program that uses MPI is executed in parallel. Pacheco describes the difference between SIMD and MIMD machines\textsuperscript{1} very simply: SIMD machines execute the code synchronously, while MIMD machines are not explicitly synchronous on the same code (or can execute different code) \cite{GeneralMIMDSystems}. Even though they share the same executable, MPI codes are executed as MIMD machines, for the reason that each processor can be at a different state in a given time. The synchronisation is not intrinsic and only occurs when communications are performed between two processes.

1.2 Existing approaches, motivations and goals

1.2.1 Project motivations

The original project proposal (see Appendix \textit{A Original Project Proposal} page 66), written by David Henty, describes how difficult it is to debug an incorrect MPI code. Several software and libraries exist on the market and provide analysis of MPI code, but most of them are complex and advanced tools. Moreover they are generally based on the generation of data while running and the real analysis is done after execution. This is not suitable for a program that does not complete, crashes or deadlocks, as it is often the case in a learning process. Therefore producing real-time information about each MPI process execution becomes helpful to whoever wants to understand why a program does not complete.

The original idea was to display a separate Graphical User Interface (GUI) for each MPI process. Each process will therefore collect and display their own information and state. Such an approach was very simple, but not optimum for the project. Chapters \textit{2 Profiling an MPI Program} and \textit{3 The Graphical User Interface} will develop further the choices and achieved functionalities.

The general motivation for the project is to provide a tool that can be used in the context of learning MPI (illustrating occurring communications, memory use, etc) and debugging (understanding why such application does not complete or behave properly). Among the several tools on the market Vampir and Scalasca are the most portable and widespread \cite{VampirScalasca} but, as we are going to see, they are not completely suited for the goals of this project.

\textsuperscript{1}Single Instruction Multiple Data and Multiple Instruction Multiple Data are part of Flynn’s taxonomy classification.
1.2.2 Visualisation tools for MPI

Communication view: VAMPIR
The Vampir (Visualisation and Analysis of MPI pRograms) tool’s goal is, as many of the largely used tools, to help the optimisation of MPI code [12]. It provides several communication views (see Figure 1.1a page 6), in order to help understand communications’ patterns. With a simple library linking and an open file format, Vampir is not complicated to use. However the actual visualisation software is available on demand and no price was found on the website. Nonetheless the file format is open, therefore other tools may be available to generate or read the traces.

Problems and behaviour: Scalasca
The Scalasca tool’s goal is more focused on the program behaviour and possible lack of parallelism than its communication patterns [13] (see Figure 1.1c page 6). It provides a set of commands that have to be used as prefix to the MPI commands². It will perform a source-to-source modification of the code, in order to add analysis tags on the compiler and the analysis when running. On the other hand the tool is free to use, but copyrighted to several contributors. It is also easily available on the Internet.

A legacy tool: XMPi
During the first part of the project another tool was found, XMpi and is one of the only tool that provided a real-time view of MPI program. Moreover it is now quite old, as its last release was in March 2008 [14]. It was developed for the LAM/MPI project, that has its last release in February 2007 [15] - it is also important to note that the LAM/MPI team is now part of the OpenMPI project. However old the GUI looks (see Figure 1.1b page 6) it provides a runtime snapshot (the real time status of a process) and other figures about the running MPI code.

While Vampir and Scalasca are very powerful tools they however both rely on post mortem analysis of a program. This is not helpful in a context where the program has a bug and does not complete. On the other hand XMpi is now getting too old to be used, as OpenMPI and MPICH2 are the current state of the art libraries.

For learning, Vampir is not a bad tool to use, as it is simply enabled with library linking. Scalasca may be more complicated to introduce to beginners that are not always from a computing background.

²A scalasca command that will run the compiler or the runner
Figure 1.1: Several tools interfaces (pictures from their respective websites).
1.2.3 Project goals

The overall project goal is to provide a usable tool to help solve common beginners problems, or possible simple bugs in the use of MPI routines. Rather than deeply analysing the code, this project aims to help solving common beginners problems. David Henty\(^3\) highlighted three main type of problems:

1. Bad communication resulting in a dead-lock
2. Sending the wrong data
3. Not waiting for asynchronous requests

Those problems cannot be solved with the previously mentioned tools, as either the execution of the program has to be completed for the analysis to operate, or tools are simply often not focused on details about the sent data. A real parallel debugger will not help a new user in getting information either, as they operate at a lower level than the MPI calls.

Therefore this project aims to generate a global view of the program, as it is executing, to help understand how it works - or does not work. The result is somewhere between a parallel debugger (as a real-time view of the program actions are displayed) and a profiling tool (with the information about the on-going communications). The goal of this project is to develop a tool which produces a real time view of the program. In that context real time means “when occurring” and will undoubtedly affect the performance of the code. The idea is to generate a teaching aid providing a simpler representation than parallel debuggers or complex analysis tools. Such a tool can help people understand how MPI communications work, by spotting typical learner errors and displaying communications effects.

The targeted programs are quite modest, running simple and typical communication on a relatively small number of nodes. A typical 2D stencil with a halo swapping on a medium-sized array (less than 100,000 elements) on 16 processors is the standard test case. In order to do so the Message Parsing Programming course case-study and coursework - along with other codes - will be used as the final codes to use (see Appendix D The test codes page 75).

From now on, the part of the project that is responsible for generating information (used as a library from the user’s perspective) will be referred as the Profiler. The Interface will be the piece of software responsible for understanding the information from the Profiler and displaying them in a graphical way.

Profiler’s goals

The primary requirement concerning the Profiler is getting information on communications (either point to point or global). The information includes timing (when did the
communication occurred), destination and the type of data sent. The data information requires a memory registration mechanism to allow the user to reference an array and will allow the Interface to display graphically what element of the array was accessed in what context (see Appendix B Project Functional Requirements page 68). Obviously a communication system has to be set up between Profiler and Interface, for the former to send information to the latter.

**Interface’s goals**

The goals for the Interface are very tightly linked to the Profiler’s, for the obvious reason that the Interface is only displaying information and not gathering it. However it has to be easy to use, for a learner to understand what it represents. To retrieve information it will be started in parallel of the Profiler, but could continue to run after the execution of this one (to allow the user to browse through the information). Exporting the generated information to a file could be a desirable feature, simply to be able to use it later.

**Project choices**

Some technical choices are very important for the project. While the original proposal was describing a tool that could work on both Fortran and C, the final project will only focus on C codes. The main reason is that Fortran bindings are done through C implementations of MPI and they are far from trivial operations. Moreover most of the MSc in HPC students used C rather than Fortran, providing potentially more users.

Providing the tool to the public is also a main choice. The tool is available on sourceforge, to whoever wants to use it. It is published under the Lesser GNU Public Licence, that is compatible with Qt and OpenMPI (that are following the LGPL as well). MPICH2 uses a US governmental licence that allows similar rights than the LGPL (reuse, copy, modify). The copyright is the author’s, Thomas Richard but EPCC is referenced. Documentation is therefore important for the project.

Finally the project aims to generate a working prototype in the time allowed (three months). Therefore efforts will be focused on functionalities explorations rather than clean and detailed design. When reaching milestones (usually when a main functionality is completed) the design and documentation will be refined, in order to provide a documented and robust tool.
Chapter 2

Profiling an MPI program

2.1 The MPI Profiling Interface

In order to allow the developers to bind to the MPI libraries the profiling interface, defined in the MPI standard, provides hooks to the normal MPI functions. The standard declares two main requirements for an implementation [2, MPI Profiling Interface: requirements]:

- Each MPI function has two entry points, one with the prefix MPI_ - that are the normal functions - and one with the prefix PMPI_. This allows the profiling library to redefine some MPI calls transparently to the user.

- The library must provide a linking mechanism that insures that non-redefined functions will be linked on the normal MPI calls.

In practice this allows any code to be build around the MPI functions, by wrapping the real calls to the MPI library - the PMPI_ ones. This approach has the advantages of being totally invisible to the user, that will call normal MPI_ functions and provide both access to parameters and return values (see Listings C.1, C.2 and C.3 page 70).

The standard does not define a specific use for the profiling interface. Even if obviously it is mainly to retrieve information, it can - and should - be used for any useful purpose [2, MPI Profiling Interface: discussion]. This a perfect definition in the case of this project, as it allows both information construction and sending of information to another process - the Interface process.

The standard approach to use the profiling interface is to develop a library that can be added to ones’ own code. Building a static or dynamic library does not matter, as the user still has to link with the MPI library to get access to the real MPI functions. The difference between static and dynamic is subtle. Static linking simply adds all the
library object files to the user’s code, while the dynamic library links with it at run time. The advantage of dynamic linking is that it avoids code duplication in several executables. While the advantage of static linking is the assurance of version compatibility (updating the library only affects one’s code if it is relinked).

For this project the choice was made to use the static library by default. The main argument is that the main cluster used - Ness - only allowed static linking with MPICH2. Moreover it turns out that the whole library is rather small, to the order of 100 KB and is therefore not a big insertion in any code (for example the main file of the MPICH2 implementation is 8.6 MB on Ness).

2.2 Profiler organisation

The profiler library (mpi_wrap) is organised in several modules, the list is available in the documentation [16]. Each module propose a set of functions to retrieve or store information of the running code. Even if it is getting the information, the profiler part of the MPI Visualisation project does not store a lot of data. Most of the time it sends data to the Interface.

- The MPI overloaded functions regroup all the MPI functions that the profiler redefines. Most of them are the basic initialisation and point to point communication functions. Some Derived datatypes are also recognised during communications (MPI_Type_vector() and MPI_Type_contiguous()).

- The user available functions are the other functions that are available to the user. They represent the other functionalities that the library proposes.

  – The first user available function is to synchronise with the user, that is to wait for a user confirmation before actually performing the MPI call.

  – Registering a communicator is also a useful tool, as non-registered communicators are ignored by the profiler and no information is sent to the Interface about them.

  – The final functionality is Array registration. This tool is used to give information about a piece of memory to the Profiler. When a communication occurs within a registered area of memory the Interface will display which elements are accessed - as long as the datatype used is known.

The other modules are for the internal use of the library only and should not be used directly by the users.

- The function tools modules regroup several functions used by other modules. They are mostly sets of individual functions that could have been in separate modules. Among them is a global variable world_rank that can be used to find the current MPI process’ rank from any where in the library.
• The template linked list is another important module. More details will be given later in the Section 2.2.2 A template linked list in C page 18.

• The Profiler-Interface protocol module is actually shared between the Profiler and the Interface. The former is in C and the latter in C++, but it does not create problems as it mostly contains value definitions. For example the Inter_message format is defined in there and represents the format of messages sent from the Profiler to the Interface.

• The Profiler-Interface functions on the other hand are used only on the Profiler side. It defines a set of functions to communicate with the Interface.

• The remaining modules are Array, Communicator and Datatypes. They define a set of functions that use template linked lists to store information about either memory, MPI communicators or MPI datatypes.

There are two types of communication occurring between the Profiler and the Interface. The first type is the simple case of data transfer: an MPI function is called and the Profiler has to send a message to the interface to give information about it. The other type occurs when a call needs to be synchronised with the user. However, in order to communicate the Profiler and Interface have to agree on a common medium and data format.

2.2.1 Communications

The medium

The original project proposal suggested a tightly coupled GUI, directly integrated in the library. The idea was based on simplicity, the graphics would be generated in the MPI calls, meaning one Interface per MPI process (see Figure 2.1a page 12). However the project aims to view the MPI program as a whole, this solution was focused on a single process. In order to have a whole view of the program, extra inter-process communications are necessary and have to be performed each time an event happen that every process needs to know about.

The second problem was more technical. Having one Interface per MPI process would require the actual GUI to be on a separate thread in order to be able to continue refreshing the window even when the program is running. This problem is not fatal, as the Model-View-Controller (MVC) approach could have allowed a modularity, with the Profiler as the Model-Controller part and the GUI as a simple View [17, Basic Concepts]. Nonetheless the result would have been several interfaces displaying the same information, either focused on a single process or (if some generalisation was performed) focused on each displaying exactly the same information about the whole program.
The approach was therefore moved from a GUI per process to a centralised Interface (see Figure 2.1b). The project could be divided in two parts: a Profiler gathering information on the MPI execution and an Interface receiving and displaying the data. Such an approach is still following a MVC, but this time with the Profiler as the Model only (it generates Data) and the Interface responsible of the Controller and View part (to the extend that the Profiler will not hold the data, as it should in typical Model object). Moreover this decoupling allows more freedom on the Interface implementation. MPI programs are usually C or Fortran code (even if C++ is provided), having an attached interface would require the GUI to be developed in the language used. Separating it allow the GUI to be developed as an external piece of software, that only handle the same type of communication. This therefore allows modularity on the Interface, to be developed with any graphical library or even to have different software for different purposes (for example having an interface to display more information about communication patterns and having one that only generates traces that could be read later).

The separation was technically possible as the MPI standard defines some functions to inter-connect two distinct MPI programs [2, Establishing Communication]. And the given reasons for such a functionality in MPI fitted the project perfectly:

1. *Two parts of an application that are started independently need to communicate.* That is the case when the Profiler and Interface are separated.

2. *A visualization tool wants to attach to a running process.*

3. *A server wants to accept connections from multiple clients.* In the client-server organisation the server owns the data, therefore in this project point of view the server would be the Profiler. To be more precise there would be several sever (one per MPI process) and only one unique client (the GUI) that retrieves information from all of them.
Using such MPI functionality was an interesting evolution of the original idea, for the main reason that the MPI_Open_port() and MPI_Comm_accept() functions provide an interconnect between two different MPI programs [2, Establishing Communication]. This also means that communication between Profiler and Interface are not directly done using sockets, but with normal MPI calls, implying the portability and availability benefits of the standard. It is important to note however, that the ports are not selected, MPI opens one and gives you the result, with no control on the actual network port that is opened.

**The data**

Now that Profiler and Interface can communicate, it is important to define a protocol they both can understand. For simplicity a string approach was taken: the Profiler sends a formatted string that the Interface can analyse. A fixed datagram (technically a C struct) could also have been used, it will be discussed here why it was not. In the code all the information about the data that both Profiler and Interface have to know are in the file inter_comm.h.

In order to differentiate the type of information sent the Profiler sends the first element of data as the message. It is a simple C enum, that tells the Interface what was the function that generated the message. For example it could be of value MESSAGE_Bsend if a call to MPI_Bsend generated the data. Each function generate its own data (some are similar, in general all the sends and receives functions generate the same information) and therefore the size of the sent information is different. This is one of the reason to use a string rather than a datagram, although padding could have been use in a datagram to have enough room for longer messages. As the size changes the Interface has to cope with this, the way it is done is explained in the Section 3.2.3 Communications with the Profiler (page 34).

Another reason is that registering a structure in MPI is quite inelegant, as each field has to be registered. More importantly it meant adding a lot of padding for the benefit of few messages (and numbers are exposed in the Appendix F Message in strings or messages in a datagram page 80). Most of the messages use little information (3 integers for a send or receive that does not imply a registered array 2.3d page 16) while other requests significantly more (6 integers for a communication that uses a registered array, Figure 2.4a; or up to 2 * N in Waitall of N requests, Figure 2.4c). Therefore the limitation of the fixed datagram did not only affect the padding that would be needed to cope with strings, but also the information messages (especially communication that uses or does not use registered memory). It would also have increase the complexity of the communications between Profiler and Interface (by sending the Waitall information in several parts for example).

**Supported functions**

The supported MPI functions are not very vast. The goal of the project was to develop a usable tool in three months rather than providing a large amount of partial function-
alities. Therefore the main effort was put on classical MPI calls, especially Point to Point that are the basis of any MPI codes. Some more MPI functions are supported nonetheless, as for example MPI_Type_vector() and MPI_Type_contiguous() to register new datatypes. The list of overloaded functions is available in the documentation [16].

Each proposed functionality (that is an overloading of an MPI function or not) sends different information to the Interface. All of them share a “basic” datagram composed of a message code to differentiate the sending function and some timing and synchronisation information. The synchronisation process will be explained later on, therefore all details here assume no user synchronisation at all.

The communication functions, explicitly MPI_Send(), MPI_Bsend(), MPI_Ssend(), MPI_ISsend(), MPI_Recv() and MPI_Irev(), send the exact same datagram (Figure 2.4a page 17). It is composed of the basic header, actually just few more information than the MPI_Wait() and MPI_Iwait() functions (Figure 2.4b page 17). Their common information is the communicator ID (comm) and the destination’s rank (dest). In the case of the receive and wait function the “destination” means the source’s rank (receiving from or waited for). The sends and receives have more information about the communication: the accessed memory; moreover the wait operation does not actually perform any direct read or write on the memory, so waiting does not require memory information. The information is the array ID, that is the identifier of a registered array (if the accessed data is in a registered piece of memory) or NO_ARRAY_ID in the other case. In the former case some more information are given, as the offsets of first and last accessed data, along with the datatype type (BASIC_DATATYPE, VECTOR_DATATYPE or CONTIGUOUS_DATATYPE as no other derived datatype are implemented).

The MPI_Waitall() function is very similar to the MPI_Wait(), to the extend that it is waiting for several requests (Figure 2.4c page 17). Therefore instead of sending one message per waited request, it sends one single message that is composed of all the information. The number of waited request sent to the interface could be different than the one given to the MPI call, as only the recognised requests are acknowledged (see the Section 2.2.3 Register MPI communicators page 20). By definition a request could belong to any communicator. Thus two arrays are sent, one destination and communicator information per request.

The other message are sent either one time (Init, Quit) or few time (registering an Array, a Datatype or a Communicator is not the most used function of a program). They send more specific information to the Interface.

The initialisation message for instance sends the process’ rank in MPI_COMM_WORLD along with the size of the same communicator (Figure 2.3a page 16).

The quit message is very simple, it only informs the Interface that this process has finished its computation (Figure 2.3b).

Registering a communicator is essential, but the sent information are relatively simple (Figure 2.3c): the identifier of the communicator (that will be used to represent the communicator in inter-communications), the number of processors in it (its size) and
the name given by the user.

Registering an array implies slightly more complicated data (Figure 2.3d). The number of dimensions of the array and the size of each one is necessary to know its number of elements - and only linear arrays are insured to work with the Profiler (see Section 2.2.4 Registering memory page 21). The name given by the user is also sent.

Registering a datatype is only implemented for vector and contiguous datatype. It implies common information for any datatype, that are its identifier (ID) and its datatype type (BASIC, VECTOR or CONTIGUOUS). Basic datatypes are classical C datatypes, like int or double and are therefore never defined by calling MPI functions. The Contiguous datatype represent the access of several elements in sequence, for example sending a row is easier to understand than sending N elements. Therefore it only adds a count value to the common information (Figure 2.3f). The Vector datatype is slightly more complex, it represent access of elements in patterns (see Figure 2.2). The registration therefore needs more information, formerly its number of repetition (count), block size and stride (Figure 2.3e).

Figure 2.2: Vector datatype information (memory and matrix view).
Figure 2.3: The registrations and initialisation messages.
Figure 2.4: The communication messages.
Communications pattern

As stated before there is two types of communication between the Profiler and the Interface. The first one is sending information to the Interface for a normal function. The other is synchronising with the Interface, *i.e.* waiting for the user to validate a communication before actually performing it. Both of them send the same information, to the extent that the latter happens before the PMPI_ call, while the former happens after (therefore time information is different). The Figure 2.5 illustrates the communication that occurs between Profiler an Interface for a call to MPI_Send() from the user’s code in both case of synchronisation or no synchronisation with the user. The synchronisation state is set up by the user, either using MPI_Wrap_set_sync() or using command line argument when starting the program (see the Section 4.2 How to use the tool page 51 for more details).

![Diagram of Communications between Profiler and Interface for a call to MPI_Ssend(). The clouds represent communications between Profiler and Interface.](image)

2.2.2 A template linked list in C

A linked list is a commonly used data structure to store information. Its main advantage is to have an adaptable size according to the amount of elements stored, while resizing an array is quite expensive (reallocation of memory is costly) and imply a huge
amount of memory to be allocated that may not be used (usually arrays are reallocated to twice their size). It is used to store information about registered communicators, datatypes and arrays.

Each element of the list is called a cell and each cell knows the next cell on the list. The first element of the list, the head, is a special cell that is used to go through the list. The tail is the last element of the list and has a link to no cell (in C a NULL pointer) and is not necessarily stored. From the head the list can be traversed element by element (giving a $O(N)$ search complexity), having the disadvantage of being costly compare to arrays, as the later can be sorted easily and each element accessed independently (with a binary search for example the search is $O(\log_2(N))$). Insertion and removal of cells is also quite costly, as cells only know their next sibling. Therefore the alternative double linked list can be used, where each cell has a next and previous link. This approach was used, in order to allow optimisation in further development and because the increase in complexity is not very important.

The Profiler needs to store several pieces of information about used communicators, registered datatypes and memory but also communication requests. Instead of implementing one linked list for every type of information, a template linked list was developed. The Template design pattern goal is to provide the skeleton of a code that does not vary and leave the sub-components define and use the parts that varies [18, Template Method]. In the case of the list of information, adding, searching and removing information does not change according to the stored data. What changes is the way data items are compared and obviously the size they occupy. As the Profiler language is C, no intrinsic Template system is available, therefore each Template object is represented as a void* pointer. In order to allow object comparison (for search purposes) a function pointer is used to reference to the appropriate function and is given at the creation of the list. The developed structure holds also information about the head and tail of the list and the number of elements (see Figure 2.6).

![Double Linked List and its Template Cells, representation.](image)

Figure 2.6: Double Linked List and its Template Cells, representation.
As the linked list is template when an object is removed it does not necessarily free the object (it could be a non-dynamically allocated variable and a `free()` on it would make the software collapse). However for convenience some `remove object` functions are proposed, that will free the the cell and its object as well. Choosing the correct one is not difficult: if the object is dynamically allocated (as it will be most of the time) and does not contain any dynamically allocated attributes (i.e. only scalar or static arrays) the `Object` methods can be used. Otherwise the simple `remove` functions have to be used and the memory liberated manually (both are shown in the Listing C.5 page 72).

It is important to note that the searching function of the linked list is very simple. It goes from the head to the tail and stops if the element is found. No mechanism is proposed to sort the list, as the simple search is fast enough. However providing a `compare` function that returns negative value for lower data, zero for identical and positive value for bigger data (as used in standard C comparison functions) could be used to perform the sorting.

### 2.2.3 Registering MPI communicators

One of the first piece of information that the Profiler has to register is the communicator. When using the MPI profiling interface, not only the user code goes through the redefined MPI function, but any other calls use it (the underlying implementation sends internal messages using it). This was found as the early stage of the project, when an enormous amount of messages was registered instead of the few the test should have generated. It also allows the user to have a view on the used communicators and focuses the visualisation on a particular type of communication (the creation of a separate communicator for a specialised type of communication is common).

The information registered about a communicator itself are very sparse: only its MPI identifier is stored along with the Profiler identifier that will be used during communications with the Interface. The template linked list is used to store the information. Each time a communication occurs, the communicator is searched into the list, element by element. The direct comparison of communicator’s ID could have been done (as most of implementation uses an `int` handle) but the `MPI_Comm_compare()` function is used for portability purposes (and looking for `MPI_IDENT`, i.e. identical communicators). This approach should not be too expensive, as the list of registered communicator should not be large (having more communicators would be beyond the scope of the project, that is aiming for a maximum of 16 MPI processes). An alternative to this could be to attach a specific attribute to a communicator [2, Caching: communicators] that contains the communicator’s entry in the list.

Registering the communicator became useful quite rapidly. For instance the request object, generated when doing a `non-blocking` communication, is an opaque datatype (an `int`). Therefore the MPI implementation - and only that - knows the details about the actual communication linked to it. It was therefore complicated to know if a particular request waited on was part of a registered communicator if no information was stored
about them. In order to do so a linked list of request information was added to the communicators’ information (see Figure 2.7). The Request_Info data-structure is composed of information about the function that generated it (a message value as mentioned in the Section 2.2.1 Communications page 11), the destination process and obviously the MPI request identifier.

Therefore every time a request is waited for a search is performed in all the registered communicators, until the request is found. This can in principle be expensive, as the number of requests can be important (the limitation in number of processes does not imply a limitation in the type of communications used) but should not be in the general case. Generally few requests are generated per iteration\(^1\) and therefore the search cost is not much higher than searching for a communicator.

![Diagram of Communicator and Request Information Data-Structures](image)

Figure 2.7: The Communicator and Request information data-structures.

2.2.4 Registering memory

Array information

Registration of memory was a key step in the project. For the main reason that prior to that only communications’ information could be displayed and only their occurrence was displayed. Therefore being able to know what part of an array was accessed during the whole program, or a single communication added a lot of possibilities.

Registering an array is done by the user, with a parameter list similar to the subarray datatype creation. The Array_register() functions takes the number of dimension of the array, the size of each dimension, the MPI datatype of each element and a pointer to the first element. The user can also specify name for the array, that will not be stored in the Profiler, but that the Interface may use as display information. From the dimension size and the datatype the whole size of the array is guessed, using MPI_Get_address() to

\(^1\)In the case of the MPP coursework, that is the biggest code used with the tool, a maximum of 8 requests could be generated per iteration (per process): a 2D decomposition implies 4 neighbours maximum, with a send and receive to them per iteration with a simple halo swapping.
get the address of the first element out of the given pointer. The Register_array data structure stores all the information (beside the name) plus an internal array identifier that will be used between the Profiler and the Interface as a reference to that array. As for communicators and requests the information is stored in a linked list.

It is important to underline that when registering an array the first element is given, its datatype and the number of elements in the array. From these information the first and last elements in memory are guessed (as addresses). Therefore when information is sent about accessed data of an array, the displacement is sent rather than the actual element’s address. This allows the Interface to have an abstract view of the array rather than knowing the actual size in memory of the array’s elements (the information will be similar than saying “accessing 12 elements from the 48th element”). This is the reason why only arrays that are linear in memory can be used.

![Linked List Diagram]

**Figure 2.8: The datatype information data-structure.**

**Datatype information**

Another important part of the memory visualisation lies in the understanding of derived datatypes. When basic datatypes (such as ints or doubles or array of them) or unknown derived datatypes are used, a single cell of the array is assumed to be used (the count attribute in the function defines how many of them). Derived datatypes are often used when the domain implies two or more dimensions. Being able to understand them at the Profiler level is important and registering information about derived datatypes is done the same way as all the other information - in a linked list.

So far only the vector and contiguous datatypes are implemented, but a modular structure was chosen to allow more to be added later (see Appendix G Functionalities enhancement: examples page 83). The profiler does not store information about the data access pattern, as only the Interface has to know exactly which memory parts were accessed or not. The Datatype_Info stores more generic information about a datatype: its identifier for communications, the basic type it is made from and a type enum that contains information about the type of datatype it is (see Figure 2.8). If the type is not BASIC_DATATYPE the pointer to the basic datatype information refers to one of the basic datatype’s information structure (it is not really used, but could be to check matching datatypes between communication and memory) otherwise it is NULL.
2.3 Limitations

The Profiler is the most important part of the project (although it is roughly 3,000 lines of C when the Interface is closer to 6,000 of C++ as it generates the information that the Interface will use. Even if not directly used by the user, with the exception of few functions, it has to be the most reliable part of the project, as both the user’s code and the Interface relies on it. However some functionalities were left for later, or partially implemented. Such functionalities are going to be presented in this section.

Timing information

The MPI visualisation project did not aim for a fast tool. Having a minimum amount of time, or at least having a known amount of time, in the Profiler is desirable. Although timing is done on the overloaded MPI functions, the “global” timing is not really performed. This is partially implemented but should be revised to contain more details, since the time purely spent in the Profiler functions and the time spent waiting for the user to validate a communication are merged, it would be appreciable that they were not. However timing information was not an important project requirement, its implementation was not finished, even in the Interface as we shall see later.

Global communications

Global communication profiling was one of the project’s major functional requirements, along with point to point and memory registration. However it was chosen to give more importance to the two other points, for the reason that point to point is more often used in beginners’ codes. Moreover the visualisation of point to point communication is the basis of the visualisation of global communications. Finally the array visualisation was very important as visualisation tool (and used in both globals and point to point). Therefore the global communications functionality was left over, for later development and finally not investigated. On the Profiler point of view it would have been a relatively easy insertion, as it would have use similar communication pattern than point to point and certainly exactly the same verification for the memory access.

More derived datatypes

As mentioned previously only the vector and contiguous datatypes were implemented. Although plenty of datatypes are defined in MPI\textsuperscript{2}, the most commonly used during the MPI code we came across during the MSc in HPC were vector, contiguous and Subarray. Vectors are used in most of the 2D decomposition codes, including the MPP coursework example (to send a column). Contiguous type is used in only one of those examples (MPPcwk_lesleis), while subarrays were usually defines for MPI-IO. Adding support for more datatypes would be useful as a visualisation tool, helping learner understand what part of memory they access.

\textsuperscript{2}vector, contiguous, Hvector, indexed, Hindexed, indexed_block, struct, subarray, darray [2, Datatypes]
Chapter 3

The Graphical User Interface

The other part of the project was to develop a graphical user interface (GUI) that will display information sent from the Profiler. A large number of libraries are available to easily implement an interface and among the most portable - and popular in the open source world - are Qt and Gtk+. The latter is developed in C, focused in GUI and is partially object oriented; while the former is a vast C++ framework not only about designing a GUI, but providing several other tools.

Decoupling the Profiler and the Interface had the main technical advantage to separate the codes. The Profiler was aiming for C, for the reasons that most of the codes available in MPI from MSc students were in C. The Interface however could gain to be developed in an Object Oriented manner, offering access to more powerful software tools. Using C++ was justified by the multiple powerful libraries available for that language, including the Standard Template Library, that could have save the implementation of the template linked list mentioned previously (Section 2.2.2 A template linked list in C page 18). Therefore Qt was chosen as the GUI library, as the author has some experience with this library and also because the framework propose functionalities that could be helpful (for example an easy preferences-saving independent of the system, or simple reading/writing of XML files with no need for another library).

3.1 The Qt framework

The Qt framework is quite huge. It contains modules, mostly oriented on GUI development, but not only (Figure 3.1 page 25). Its two mains components are the QtCore and.QtGui, that provides the basic classes for all Qt application and the user interface classes (the 2D graphics is a separate module that handles direct drawing on an area). However some more modules are available, for example to integrate OpenGL code in the graphics, use XML simply or communicate over a Network to only mention a few of them.
This section will focus on some elements of the framework that are used in the project. It assumes some knowledge about Object Oriented Programming (OOP) and some basic C++ understanding. The concepts explained here are not required to understand the Interface organisation, but they are the basis of the GUI development.

### 3.1.1 Qt root: the QObject class

Qt relies on the Object Oriented features that are Inheritance and Polymorphism. The first is used to derived a new class from an existing one and keep its properties, while the other allows several inheritance to be done to create a tree of child classes with different features. In Qt all object inherits from QObject and the class provides the basic mechanisms, for example the signals and slots ones [19, QObject].

**Object tree**

All classes inheriting from QObject can be organised in object trees. This is a mechanism by which QObject defines parental links between objects and by giving the responsibility to the parents to free their direct children in their destructor. This creates a destruction cascade when one of the top object’s destructor is called.

This is a very important feature of Qt, that ensures that QObject based children will be freed automatically. This explains too why there is little memory freeing in the Interface code, as most of it is handle by the Object tree mechanism.

**Copy and assignment**

In C++ the canonical class is a class that provides at least some specific methods, that are the base of the OOP techniques [20]:

- default constructor (no parameters or with default parameters)
- copy constructor (construction from instance of the same class)
- virtual destructor (to allow inheritance)
• assignment operator (the operator= is used to copy values of two different instances of the same class)

• any other meaningful operator (for example a Position class could define an addition operator operator+, or the two array subscripting operator[]).

The QObject provides all of that, to the extend that a design choice made the copy constructor and the assignment operator private, hence inaccessible to children classes. The reasons are discussed in the Qt documentation [19, QObject: Identity vs Value].

**Signals and slots**

In simple GUI libraries, like OpenGL, a particular mechanism is set up to handle events. Events are organised in types, for example drawing the screen and user actions. The user writes some functions that are linked to each event type, that are the *callbacks*. In the callback the event is analysed, usually in a switch or an if’s cascade, in order to make the software react.

```cpp
switch (action)
{
    case MOUSE_CLICK:
        if (subaction == LEFT_CLICK) // do left click action
        else if (subaction == RIGHT_CLICK) // do right click action
            break;
}
```

Listing 3.1: Pseudo callback switch.

The signals and slots mechanism is developed to provide an alternative to method callbacks. Instead of linking events to statics methods, signals can be *connected* to other methods, the slots, of the same object or of other objects. When the event occurs the signal is *emitted* from the source object and an underlying mechanism calls each connected slot. To use this mechanism, the class must inherit from QObject (directly or indirectly through one of its children) and insert the `Q_OBJECT` macro previous to any other attribute or method [19, Signals & Slots].

Signals are special methods that must not be implemented and can never have a returned type. The are emitted using the *emit* keyword, or by normal invocation (but the former is preferable for readability). The `Q_OBJECT` macro allows the use of the *signals* visibility that allows the class to define signals (like *public*, *private* or *protected* visibility statements).

It is important to note that slots are normal C++ methods, that can be called without signals linked to them, they only need to be referred to as *slots* in the visibility declaration (example: *public slots*). However their visibility to other classes is slightly different. Using normal invocation they follow their C++ visibility (slots can be *public*, *private* or *protected*) but as callback functions they are always considered *public* as any other class can attach signals to them.
Several slots can be linked to a single signal and vice versa. Therefore they are a useful mechanism to update objects with no direct calls in the code, replacing the event switch that can be found in other libraries. Slots can also be declared as virtual and child classes can redefine them. Their only limitation may be the run-time linking, that is slower than normal invocation. Moreover the connection between signals and slots is only partially checked at compilation time (only type matching is done, checking that the signal emits the same parameters than the slots receives) but their actual existence in the objects is verified at run time.

3.1.2 User Interface and Graphics in Qt

The GUI classes in Qt are divided into two groups: the Widget based ones and the Graphics based ones. The two trees are not mutually exclusive, but are not based on the same principle.

The Widget classes represents the main components of the GUI in Qt. They are all based on QWidget, that is a basic, complete, atomic element of a user interface. By definition a widget could be anything, from a button to a window as it provides the basic functions of the UI. The QWidget therefore provides methods to handle user interaction events (like clicks or resize actions), drawing mechanisms, windows and dialogues specific options. The QObject part of it allows signals and slots mechanisms along with the object tree organisation.

The Graphics tree is another way of displaying graphics. Rather than focusing on components (buttons, windows) it focuses on items (shapes, lines) and display options for them (zoom, rotation, translation). It is closer to a lower level of drawing, organised in scenes (using the QGraphicsScene object). The scenes are displayed into a view (QGraphicsView objects), that is the link between the graphics and widget trees.

It is nonetheless possible to display a Widget based object inside a graphics scene. This method uses a proxy (QGraphicsProxyWidget objects) that allows a normal widget to become part of a graphics scene, with the inconvenience of some performance drawback if complex image modification or widgets are used.

3.1.3 Threading environment

Qt provides its own abstract thread class, QThread, that can be inherited. The actual library used for the threads is wrapped, so the user does not have to worry about portability and more importantly the signals and slots mechanism works between QThreads. When a thread is started it executes its abstract method run(), that has to be redefined by child classes. However if signals and slots have to be used the actual code should not be directly inserted into the run() method. It is due to the fact that the threads have to run the exec() method (from QObject) in order to attach signals and slots at run time.
What can be done is a simple trick, by linking the `started()` signal to a slot that will indeed perform the thread’s actions (see Figure 3.2).

The GUI’s thread is independent of the other threads and the user does not have to define threads for the GUI in normal cases. QThreads are provided to have alternative threads that performs the expensive computation without affecting the interface refreshing rate.

### 3.1.4 Other tools of the Qt SDK

The Qt SDK provides several tools that aim to help the developers building and deploying their code as easily as possible.

**QMake**

A Qt project is generally built using project files (.pro) that contains information about the project itself. Not only the source and header files are listed, but also the Qt forms (Qt Designer .ui files), some external resources (icons, pictures, etc), name of the executable or library to generate and libraries dependencies to mention some used in this project. The `qmake` tool analyses the project file and build one or several Makefile accordingly. This allows the project’s writers to focus on information rather than technical details (doing a multi folded project is quite easy with QMake for example).
In the case of this project information are given about the MPI library (that is used for communication between Profiler and Interface) as any normal external library. The path to the installation folder is also provided, along with the path to the Profiler’s header files needed for the Interface (inter_comm.h contains all the Inter-communication information needed and is defined by the Profiler part of the project).

**Qt Designer: create a GUI easily**

Among the tools provided by the Qt SDK is the Qt Designer. This tool allows simple creation of User Interfaces (UI) that can later be used directly from the sources. The designer itself produces a specialised XML file with the extension .ui, that the uic compiler (provided by the Qt SDK as well) transforms into a header file containing a class definition in a `Ui`: C++ package. This class contains a `setupUi(QBaseObject*)` method that will populate a given base object (any of the QtGui classes, like QWidget or QPushButton) to create the UI as shown in the designer.

The Qt documentation defines three ways of using `Ui`: classes, but this project only uses the most complete one: the *Multiple inheritance Approach* [19, Qt Designer: Using a Designer UI file in Your Application]. Each UI developed in the designer is based on a base class, usually QWidget or one of its direct children (for example QDialog is used for the Starter class). The multiple inheritance approach is based on the inheritance of both the base class and the UI class. This allows full control over the signals and slots, but also on the UI components. Each class of the Interface has a private `init()` method that is used to setup the UI (either using the inherited `setupUi()` if it is based on a designer class, or to initialise its attributes). Usually the signals and slots connections are done in that function too.

**Qt Linguist**

Qt provides a very simple way to translate an application, using the QObject translate method to generate the messages (QObject::tr()). This method simply return a string if the Linguist is not used. When the Linguist tool is used it allows the interface to be easily translated in dozen of languages (even in languages such as Arabic, that needs a right to left writing). The language of the interface can be changes at both compile and run time, for the latter needs implementation of special signals and slots, usually called `retranslateUi()` for example in the Qt Designer UI classes, that will be triggered when the user changes the language.

The Qt Linguist tool is not used in this project, but could be in the future. All of the messages displayed in the interface (the logs are excluded) are generated using the translate system. No re-translate method was developed, but such a modification is trivial.
3.2 The Interface organisation

The Interface code is a substantial part of the project, in size it is twice the Profiler and it is directly used by the user. It was driven by an experiment goal, trying to introduce new features that may be useful. Therefore its organisation is not as clearly defined as the Profiler’s one and the code is far less commented. The code proposed for this project should therefore be considered as a prototype, working but not extremely well designed. As it will be detailed in the Limitations (Section 3.3 Limitations page 46) re-factoring the project is going to be needed in the future.

The Interface is developed using Qt, therefore it is using C++. The only class that uses MPI directly is the monitor between the Profiler and the Interface (MPIWatch) and is therefore using the C++ binding of MPI. Main implementations usually have such binding (as it is defined in the standard) but generally lack of proper documentation especially on the classes organisation (the documentation exists, as the functionalities are the same than their C homologue, but can only be found from the C version).

As a reminder, it was explained before how the Profiler is communicating with the Interface: using standard MPI calls within a special inter-communicator. Therefore this section will not only focus on the class organisation, but also on how the messages are analysed and stored in the Interface. It will also detail more the functionalities of the Interface that are the memory display and the synchronisation with the user. Finally it will discuss the limitations of the Interface and some possible future improvements.

3.2.1 The GUI organisation

Main classes

As shown in the Figure 3.3 (page 31) the GUI is centralised on the MainWindow class. It is the point of entry of the Interface, to the exception of the Starter class, that is used to retrieve the Profiler’s connection information (see further in Section 3.2.2 Connection to the Profiler page 32). The main window is composed of several sub-windows, that are either tabular elements or dock elements. The formers are arranged in tabs, that can be move around but not closed, while the latter are former sub-windows: they can be moved in or out the main window and the user arranged them in the main window freely (as the docks are lists the top and bottom space are deactivated) [19, QMainWindow].

The tabular-windows contains the main information of the Interface:

- The Statistics that contains information about each registered MPI communicator and that will be detailed in the Section 3.2.4 Memory visualisation Statistics (page 38).
- The Communication visualisation that are one tab per registered communicator. They contain the view for the communications to occur in, composed of processors and links between them. It will be detailed into the Section 3.2.5 Communication visualisation (page 41).
The dock-windows are mostly used to display more information to the user. They are accessible through the menu view and are generally compulsory to display.

- The Datatype View contains all the registered datatypes and displays them. A new datatype is registered in it if it has a different memory access pattern, it means that two identical vectors will only have one entry, even if they are not based on the same datatype (for the Interface the memory is considered as cells, that can be any size). The behaviour and the look of this dock is essentially the same than the one described in the Section 3.2.6 Memory visualisation (page 43), as it is using the same classes (ArrayView and Datatype).

- The Requests List is the dock window that may not be compulsory to display. It displays the list of MPI calls that are asked to be synchronised with the user (i.e. requires a mouse click to progress) and is automatically displayed when receiving a new request. More information will be given in the Section 3.2.5 Communication visualisation (page 41).

![Figure 3.3: The Interface classes organisation.](image-url)
Other classes

Some classes are nonetheless not shown in Figure 3.3 (page 31), for the reason that they are either used in most of the classes, or used as information holders.

- The *Exception* class was developed for the author’s Honours project\(^1\), that was also using Qt. It is a class that holds information about an exception (a message and if it a fatal error or just a warning) along with the possibility of being displayed on standard outputs or using Qt pop-up messages `QMessageBox`.

- The *Log* class is used to write on the standard output for either informative or debug purpose. It is a wrapper to the standard error output, with a mutual exclusive access lock (mutex) to prevent multiple threads to write on it at the same time.

- The *MessageInfo* class is used to store information about a received message from the Profiler. It is used as a transfer class that is copied from objects to objects using signals and slots, as it will be detailed in the Section 3.2.3 *Communications with the Profiler* (page 34).

- The *Starter* is the dialogue window shown to the user just before starting the Interface. It allows the user to choose the way of connection with the Profiler. It will be detailed in Section 3.2.2 *Connection to the Profiler* (page 32).

From the above description it is obvious that every class knows the *Log* object, that is in fact implemented as a Singleton design pattern. The Singleton allows any object to access a unique element and insures that this element is unique [18, Singleton]. In the case of a multi-threaded application (that is the case of this project) the Singleton elements have to be protected by a mutex.

3.2.2 Connection to the Profiler

Remember, the Profiler opens several ports automatically (one per MPI process) that the Interface has to connect on. As a matter of fact the Interface therefore needs the ports and it is the duty of the user to provide the ports to the Interface. It is nonetheless not a complicated task, as the Profiler implements several ways of publishing the ports (see details in the Section 4.2 *How to use the tool* page 51). In order to cope with all the ways, the Interface has a special dialogue that is shown before it actually starts (see Figure 3.4 page 33). This dialogue will allow the user to either type, or copy, each port manually but it will also allow the selection of a file where one port is stored per line. The latter technique is the easiest to perform, as the Profiler proposes a `--port-in-file` command line option.

The number of MPI processes is either guessed from the number of lines of the port file, or manually selected by the user. No port checking is done, as different implementations have different formats (see Table 3.1 page 33 for the differences between MPICH2 and OpenMPI), but the number of found ports is checked against the number of processes.

\(^1\)3D Surface Data Editor, for the Robert Gordon University

32
It is also important to note that the file option uses MPI-IO on the Profiler side. It implies that the file is not erased prior to writing. It means that a reused file that, for example, used to have 8 processes in it and now will have 4, will only replace the 4 first lines. This may trick the Interface, that finds 8 lines and 8 ports, when only 4 are relevant. The problem can be avoid by always deleting the file before exporting into it, or by manually adjusting the number of processors.

<table>
<thead>
<tr>
<th>rank</th>
<th>MPICH2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>tag=0 port=59396 description=ness.epcc.ed.ac.uk ifname=129.215.175.1</td>
</tr>
<tr>
<td>1</td>
<td>tag=0 port=53293 description=ness.epcc.ed.ac.uk ifname=129.215.175.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rank</th>
<th>OpenMPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>946733056.0;tcp://192.168.1.2:51236+946733057.0;tcp://192.168.1.2:56069:300</td>
</tr>
<tr>
<td>1</td>
<td>946733056.0;tcp://192.168.1.2:51236+946733057.1;tcp://192.168.1.2:57533:300</td>
</tr>
</tbody>
</table>

Table 3.1: Different port formatting for MPICH2 and OpenMPI.

**Actual connection**

The actual connection is done when the Interface is really started (i.e. when the main window is created). The main window creates one MPIWatch object per MPI process and gives them their port. The MPIWatch objects are monitors for communication between the Profiler and the Interface (see Section 3.2.3 Connection to the Profiler) and they connect to the Profiler part before starting their thread.

This implies that the Interface is hidden while the connection is done and remains hidden until all monitors are connected. The problem for that design is the inability for the user to know if the connection failed or if it is just long (the project was tested on a Shared Memory cluster, making the connection almost instantaneous, but it may not
be the case for other type of clusters). An alternative way was experimented, but is not used any more. A separate window was created for the connection, with a time out running in it, to be destroyed when all the connection occurred. However this implied the monitors to be connected after their thread started (to avoid the GUI to freeze in the blocking call of MPI::Intracomm::Connect()) and implied too much re factoring according to the time left\(^2\). Remnants of this can be found in the code, as the Connection class is available, in order to be used in the future.

### 3.2.3 Communications with the Profiler

As stated before, the MPIWatch is the monitor object that receives information from the Profiler. It actually delegate the parsing to the MessageInfo class, that will be also be used to share the information with other classes.

**MPIWatch**

The MPIWatch class is the only class of the Interface that uses MPI. It connects to the Profiler using the MPI::Intracomm::Connect() method, that connects two MPI::Intracomm together (from different applications) and return an MPI::Intercomm object. In order to save the refreshment rate it is started on another thread, using the previously explained QThread inheritance (see Section 3.1.3 Threading environment page 27).

The Figure 3.5c (page 35) shows that it continuously does synchronous receives from the Profiler, until the Quit message is sent. Then it simply stops its own thread. As previously explained (see Section 2.2.1 Communications page 11), the messages sent from the Profiler to the Interface are not constant in size. In order to cope with that the monitor tests the next incoming MPI message and check its size, using MPI::Comm::Probe(), before performing the actual receive. An allocated buffer is used to receive the messages and its size is increased if needed.

The signal emitted from the Profiler is different according to the received message. This could not have been the case if just send the MessageInfo object to every object that needed information and delegating the task of choosing if the message was relevant to the linked objects. Instead of that the MPIWatch object defines different signals for different messages to help filtering. The name of the signals are self-explanatory:

- `newCommunicator`
- `newArray`
- `newDatatype`
- `newMessage`

\(^2\)The simple implementation tried resulted in unstable connection. Some threads did not connect correctly, as a race was occurring between the GUI thread and the MPIWatch thread to set up the signals and slots communications.
(a) Acknowledge a user action: sendMessage().

(b) Wait on the requests: waitForRequests().

(c) Receive from the Profiler: monitor() method.

Figure 3.5: MPIWatch communications with the Profiler.
Finally the MPIWatch objects are also responsible to send the acknowledge message, used to inform the Profiler that the user validate a blocked message. Remember that the user has to have the ability to stop the execution of chosen MPI calls, in order to have the time to see the accessed memory, or simple to get more information about the on-going communications. This is achieved by proposing the user to click on a `continue` button and the actual MPI call will be delayed until this action is performed. Only one message can be synchronised with the user per MPI process at one time as an active wait is done (waiting for the acknowledgement from the Interface, as shown in previous Figure 2.5 page 18). Moreover the actual content of the message sent is irrelevant (the Profiler simply does a `MPI_Recv()` without reading what it receives).

In early version the method was performing a `MPI::Comm::Send()` directly from the MPIWatch::sendMessage() method. The latter was called from the MessageInfo::acknowledge() method, triggered when the user clicked the `continue` button. This introduced problems if the Profiler stopped for some reasons, or the communication is delayed for other reasons. Actually, even if the monitor is on another thread, the method is called from the display thread, therefore delaying that MPI call is delaying the GUI refreshment.

Therefore the acknowledgement mechanism was changed. A mutex was added to the MPIWatch objects along with a MPI request list. When the GUI thread sends its acknowledgement it now does it with `MPI::Comm::Issend()`, including no more delays. The request is then waited on the next loop turn of the MPIWatch::monitor() method. The population and waiting of the requests is displayed in the Figure 3.5 (page 35). Note that a different tag is used in the communication for the messages from the Profiler (information messages) and the messages from the Interface (acknowledgement messages) in order to avoid confusion.

**MessageInfo**

The MessageInfo class is responsible for understanding the messages from the Profiler (see Figure 3.6 page 37). It therefore proposes a parsing method, that the MPIWatch objects will call when receiving messages in order to extract the information out of them. It is also used to carry information from one object to another in the Interface. It is sent from the monitors to any other classes using signals and slots and therefore is implemented as a canonical class (allowing copy constructions and affection operators to be used).

Not all its fields are used for any particular message and the Table 3.2 (page 37) lists the different usages. Note that the attribute `elems` is a dynamically allocated array, that therefore requires special care in the copy construction, affectation operation and the destruction of the MessageInfo objects.

The MessageInfo object is also used to generate the Datatype object when registering a new datatype and, as stated before, to acknowledge synchronous messages to the Profiler.
Figure 3.6: The MessageInfo class diagram.

| _message | message enum value |
| _time_in | time in the function |
| time_out | message specific |
| u_synch | user synchronisation |
(a) All messages usage

| _comm | communicator’s id |
| _size | communicator’s size |
| info | communicator’s name |
(c) Register Communicator

| _datatype_id | datatype id |
| _first_elem | datatype type value |

Vectors:

| _size | 3 |
| elems [0] | count |
| elems [1] | block length |
| elems [2] | size |

Contiguous:

| _size | count |
(e) Register Datatype

| _first_elem | number of waited |
| _size | 2*_first_elem |
| elems [0 : _first_elem] | communicators’ id |
| elems [_first_elem : _size] | processors’ id |
(b) Waitall

| _array_id | array identifier |
| _size | number of dimensions |
| elems [0 : _size] | size of each dimension |
| info | array’s name |
(d) Register Array

| time_out | time out of the MPI function |
| _comm | affected communicator |
| _to | destination/source processor’s id |
| _array_id | array identifier |

If array_id is not NO_ARRAY_ID:

| _first_elem | first accessed element |
| _size | last accessed element |
| _datatype | datatype id |
(f) Ssend, Bsend, Issend, Recv, Irecv and Wait

Table 3.2: The different attribute usage of the MessageInfo according to the message.
See Figures 2.3 and 2.4 (page 16 and 17) for the matching datagrams.


**Propagation of MessageInfo**

The MessageInfo data-structure is emitted from the MPIWatch objects and used in several classes as source of information. The connection, however, is not direct between the monitor and the final objects. The messages are in fact sent to the main classes (Statistics, CommunicatorsTab, RequestsList and DatatypeView) to be then distributed to the right objects (Figure 3.7 page 39).

The most sent signal is newMessage(), for the obvious reason that most of the communication between the Profiler and the Interface is information about MPI calls. The propagation through the Statistics and CommunicatorsTab is usually done to select the correct communicator, while the RequestsList checks if the communication requests a synchronisation with the user (if it does a new Request object is created).

The other signals are used less often and all of them imply the creation of objects. Registering a new communicator is the signals that implies the most object creation, as the Statistics and CommunicatorsTab will have to populate their data-structures with new processors information. The longest copy-link is done through the CommunicatorsTab and in the future organising the data storage could help reducing this copy cost, memory wise (stack calls mainly) as well as time wise as signals and slots connection is slower than direct methods call.

3.2.4 **Statistics**

It is important to provide the user with information about the running MPI code. These information nonetheless have to serve a purpose and be as evident to use as possible as, it is good to remember, the target user is a beginner in MPI programming. The statistics are therefore composed of three types of numbers. The first one is the calls statistics, that is the number of call to a given MPI function. The second one is timing statistics, that is incomplete, but should represent time spent in a specific MPI function, as well as time in the overall MPI program and in the Profiler’s function too. The final one is communication information, that is meta-knowledge about the MPI program. It displays the number of sends and receives done, independently of the type of send/receive action used (MPI_Issend or MPI_Ssend are sending functions), the number of global communications performed and the number of generated and waited requests. This last statistics are certainly the most important for the learner that wants to understand why one code may or may not complete correctly.

The user can change from one statistics to another using the menu on left hand-side of the panel and each communicator has its own set of statistics (see Figure 3.8 page 40).

More technically the Statistics class is a mere display, that handles the choice of communicator and information to display. Information is stored in the CommInformation objects that the Statistics create when a communicator is create. As the whole Interface was designed as a prototype few efforts were made to organise the information.
Figure 3.7: The several MPIWatch signals propagation.
Figure 3.8: The three statistics information windows.
The actual implementation does not store arrays of data, but directly the QTableWidget objects. This was done for simplicity, as storing raw data would have implied an update mechanism to be set up. Therefore the CommInformation is a temporary class, that will be replaced in the future to have more distinct separation of GUI/data-storage.

The overall idea of the Interface is to store data where needed. In the future a Data:: and Display:: namespaces will be create. This will allow a MVC approach to be applied on the Interface, allowing both better performance (will certainly reduce memory usage) and readability of the code (see Section 3.3 Limitations page 46).

3.2.5 Communications visualisation

The communication visualisation regroups the communications display (who is communicating with who). It is also responsible to handle the synchronisation with the user, showing him that communications need validation and allowing the user to validate them. It is displayed in the Communications tabbed window. This window is itself composed of one tab per registered communicator.

For each process in the communicator a small widget is created, from the Processor class, that has basic information about the process. Its name (a mixture of its communicator’s name and its rank) is displayed in the title bar, but also in a separate field. It is due to the fact that Qt’s display look changes from a platform to another and sometime title-bar text is difficult to read\(^3\). If arrays are registered they are displayed in the Processor’s widget, in their own box (see Section 3.2.6 Memory visualisation page 43). The Processor’s latest status is also displayed, or the on-going status if the communication is waiting for user synchronisation. The option button (with a label of “…” and an arrow for the menu) allows the user to set few options about one processor, like the name display, which array to show and also some options about the array display.

Note that the user can move and resize the Processor’s box at will and rearrange them in different positions according to the communication patterns. The links between processors will behave as expected: they will stay attached to their processors. This may be problematic on a large number of processors, but as the project aims for a reasonable number of processes (16) the results are correct.

Normal communications

Normal communication are not synchronised with the user. When two communicators communicate a line is drawn between them. If the communication is occurring the line is black and grey if it occurred in the past. The Figure 3.9a (page 42) is an example of this view.

\(^3\)The differences of look between Windows, OS X and GNU/Linux are obvious, but even between X11 libraries used for user interface in Linux could have different rendering.
Figure 3.9: An occurring MPI_Ssend() synchronised with the user.
Synchronise with the user

If the communication is synchronised with the user a Request object will be created and added to the RequestList (see Figure 3.9b page 42). The RequestList is a movable dock window that provides the list of requests, as long as options to deal with them. Each request as information about the communication, like the destination and the time it occurred, that can be hidden or displayed with the show/hide button (the + or – button). The continue button validates the communication and a small ball will travel the link between the two processors before setting it to a grey colour.

The request list also provides a continue all button, that validates all of the available requests and an auto continue option. The auto-continue is a timer that will validate the first request in the list when its time is 0 and allows the user to see the messages occurring in sequence without having to regularly click on buttons to validate them.

3.2.6 Memory visualisation

Memory visualisation was a key step for the Interface. While registering an array is relatively simple on the Profiler’s side, the Interface has to cope with the multiple possibilities of the number of dimension and size of the array. The display of any size and number of dimension is possible, although the project focused on linear and 2D arrays. It was also important to be able to access each array’s element, in order to display which one is accessed during communications.

The memory visualisation is used for both the Datatype View and the array access display (in the Processors). In both cases it uses the ArrayView class to display each element of the array as a whole (1D is linear array and 2D as a Matrix). It is a simple list of every element of the array, called cells, that is organised in a convenient display manner. The colour of each cell represents its status, there is five possible status for one cell:

- Never used (white)
- Currently read (red)
- Previously read (yellow)
- Currently written (blue)
- Previously written (grey)

For the array visualisation in the Processor each cell is represented as two triangles, the upper ones displays the reading status, while the lower one displays the writing status. Another approach would have been to do colour blending, to have instead of the five current status seven of them but it was chosen to have separate elements for readability.

In matrix view the memory is displayed as C order, meaning that a [3][10] matrix is a 3 rows, 10 columns matrix. The upper row is the first row. Switching view is possible

\footnote{Creating 2\(^3\) status (never, currently, past), by combination 8 status, but 2 are equivalent.}
from the option button’s menu when the number of dimension is appropriate (if there is more than 2 dimensions the first dimension is used as the base, creating large rows for the other dimensions) as shown in Figure 3.10). For example the previously mention figure could either represent the [3][10] matrix or a [3][5][2] matrix.

The Datatype data-structure is also used to store information about a datatype. It is an abstract class, that could not directly be implemented. The Vector_Datatype and Contiguous_Datatype are subclasses of Datatype, static casts are done at run time by checking the type of the used datatype (see Figure 3.11). Any new datatype has to redefine the access() method, that should give which elements were accessed, according to the first accessed and the number of datatype used.

(a) Matrix view  (b) Memory view

Figure 3.10: A vector datatype memory and matrix view.

```
Datatype
| id : uint |
| <enum>   | type : Datatype_type |
| operator=: (in other : Datatype, out result : bool) |
| access(out result : int [*], in first : int, in count : int) |
```

```
Vector_Datatype
| count : int |
| blocklength : int |
| stride : int |
| access() |
```

```
Contiguous_Datatype
| count : int |
| access() |
```

Figure 3.11: The Datatype classes tree.
3.2.7 Export information

Exporting information gathered from the Profiler is one of the latest functionality developed. It was important to do it, in order to set up the basis for further exportation. The implementation generates a snapshot of the current values held in the Interface. The file is formatted using XML, selected for simplicity of future conversion and also because Qt provides basic classes to read and write XML easily. The XML tree generated is exposed in Figure 3.12.

Not all the information from the Interface are exported, for instance all the display options are not. The statistics, the processors’ links and their registered data is exported nonetheless, the read and written cells of the arrays are also exported. No information is exported on the registered datatypes. This was chosen as the XML file could be used to view the final stage. However any data could be added in the future, quite easily.

Figure 3.12: The XML information tree. No repetition values means 1.
3.3 Limitations

Design

The actual Interface organisation is a bit messy. The idea was to develop a working prototype that could be refined later if the project proposed interesting ideas. The point where the tool provides enough features to be used is almost complete, as only global operations are really missing from the display.

A new design should separate even more the communications, data storage and display. The Profiler would send information to classes in a Data:: namespace, while the Display:: namespace classes would handle UI with Qt (see Figure 3.13). The Data:: classes would be responsible for storing the information, but also to inform the Display classes. This design would allow more complex information analysis from the messages as well, as grouped information would be simpler to analyse. This could also save some memory, although not much information is repeated in the software (datatypes information might be the only ones and they are fairly light).

Figure 3.13: The MVC organisation design.

Timing

Timing information, for the same reasons than the Profiler, was not totally implemented. Some more data could have been sent from the Profiler and used, but also some data sent from the Profiler are not really used. As the global time sent when quitting is not exact, it is not used to computed the time spent into the Profiler functions (one of the empty fields of the Time statistics). Some other interesting functionalities could have been to save the starting time, and display occurring actions as a displacement from it (having 1 sec displayed instead of a mystical $1.31e+9$).
Memory visualisation: memory glutton

The actual implementation of the Interface is using an important amount of memory. The numbers will be exposed in the Section 4.3 Applications results (page 55), but for the small test programs the memory used was to the order of 300 MB, while reaching several GB for the coursework. The source of the memory need of the program is not mystical, it comes from the communication view. For simplicity reasons, QWidgets were embedded into QGraphicsScenes in order to have normal UI widgets movable in a free view (the processors can be moves). However a second level of QGraphicsScene is then needed to display each registered array, creating some complex storage on the Qt side. Moreover each cell of an array is composed of two triangles. Although Qt proposes some direct drawing methods (drawing shapes) it is not recommended to use this library when either the animations are complex, or the amount of drawing is important. In the case of a 100,000 elements arrays, this implies 200,000 triangles per processor, going further than the million elements for more than 4 processes.

Alternative way could be taken in the future. Using a redesigned Interface using MVC, a limited view of the array could be done, implementing the scrolling instead of give all the elements to the view. Or alternative libraries could be used to display the array information. OpenGL could be a decent choice as an alternative library, in order to display 3D arrays and also because it is easily integrable into Qt.
Chapter 4

Achievements and Future work

For a project that ran as long as the one presented here, it is important to present and discuss the current results. Therefore the reader should remember the original goals of the project: producing a teaching tool for beginner users and the functional requirements list (see Appendix B Project Functional Requirements page 68).

4.1 Profiling usages

This section will detail some of the possible profiling usages achievable with the current tool. It will for instance give some examples of mistakes that could be spotted with the tool.

4.1.1 Technical usage

Ssend and Recv deadlock

The simple message in a ring example (Listing C.6 page 73) is certainly one of the first codes a beginner will write in MPI. However it may not be clear to oneself yet why a non-blocking communication should be used, therefore one can easy perform a MPI_Ssend() follower by a MPI_Recv(), that is an obvious case of deadlock.

To spot this deadlock with the tool is relatively simple, but not trivial. As the Interface is a prototype it does not really “analyse” information, it just displays it. No information is given directly to the user that the MPI call was performed or not. Therefore when a user wants to know why one’s code deadlocks, they should run in synchronised mode for the functions that may deadlock (in the case of messages in a ring, MPI_Ssend() and MPI_Recv() are the two functions used). Although the deduction could be done on simple code by simply looking at the latest status displayed (in this example MPI_Ssend()). While the code executes the user validates communication per communication and will
see what was the latest call was before deadlock occurs, consequently the problem may be deducted.

**Bad halo swapping**

In order to visualise halo swapping the user has to register meaningful arrays. A bad halo swapping example is provided for a 1D array and will be extended to 2D visualisation further.

The traffic example (see Appendix D *The test codes* page 75) is provided in two versions or 1D halo swapping. One will perform bad halo swapping, one will perform the correct halo swapping. The Figure 4.1 displays both of them, on an array of size 10. The halos are two elements added as copies of the previous and next cells’ values (see Figure 4.1a). Therefore the communications occurring in the Figure 4.1b are obviously wrong. The halo data is sent and received, but it should only be received. Correct halo swapping is displayed in Figure 4.1c: last meaningful data is sent (read) while the halo are only received.

![Figure 4.1: The correct and wrong halo swapping for the traffic model.](image)

The visualisation of 2D arrays was very important for the project. Most of the scientific codes uses 2D, or 3D decomposition of a problem - either to represent a spatial information or simply because splitting matrix information is common. Using MPI, one line of an array is straightforward to send: only the *count* value should change (to be set as N, for a row of size N). However using a column requires the usage of a vector datatype.

In order to test the vector datatype the *matrix* test case was developed. The code simply perform a matrix transposition. The Figure 4.2 (page 50) illustrate the communication when sending one, two or five vectors during one communication for a 10x10 square matrix. The visualisation is exactly the same as single data sending and same errors could therefore be spotted. In first version the rows were vectors, they are now Contiguous datatypes, in order to test this datatype as well.
Figure 4.2: Matrix transposition visualisation. Red cells are sent into the blue cells.

Not waiting on requests
Not waiting on requests is a common mistake, that is easily avoidable by looking at the communications information in the statistics windows. It is detailed later, in the example on how to use the tool.

Derived Datatype verification
With the visualisation of derived datatype it is easy for a learner to understand the effect of a datatype on a 2D or linear array. The options to visualise the datatype can help, for example when defining vectors and in the future other derived datatypes support will enhance this possibility. This visualisation was detailed in the Section 3.2.6 Memory visualisation (page 43).

Memory access pattern
When an array is registered every overloaded function that uses it will send information to the Interface. It will then display which part was read and written during the whole program. It is therefore useful to see what array is used for the communications. In a context of multi-communicators it is also interesting to see that the arrays may be used differently on different communicators (in the case of sub-communicators performing local operation on rows and columns, as used for example in Fast Fourier Transforms). This functionality is used on the MPPcwk_paul version to show the deep halo arrays accesses (see Section 4.3.2 The MPP coursework page 57).
4.2 How to use the tool

Now that the reader knows more about what the tool could be used for, it becomes interesting to show how easy it is to use it. It is important to note that the MPI Visualisation deliverables are a library, mpi_wrap and an executable, mpidisplay. The usage can be divided in five steps.

1. Modify the original MPI source code to include the library (mpi_wrap).
2. Run the code (with the Profiler), that generates the MPI ports.
3. Run the Interface (mpidisplay) and connect to the MPI ports.
4. Visualisation in real time.
5. The MPI code completes.

The first step is due to the fact that the user has to add library linking, in order to include the Profiler to the code. This will make any call to a redefined MPI function go through the Profiler, send a message to Interface and perform the classical MPI code (see previous Figure 2.5 page 18). While synchronisation with the user could be set up at run time, some information could - and has to - be added to the code. This information concerns the MPI communicators to registers and also the arrays to reference.

Running the code is done using the same way as any normal MPI code, but command line arguments can be given. The most important one concerns the port publication, in order to select what media to use (standard output or file). Some others are also provided to synchronise with the user when a particular MPI call is performed. The Interface is started as another process, using the executable mpidisplay executable. The visualisation is then performed in real time, the Interface receiving and display the data as soon as the Profiler sent them. Eventually the MPI code will complete, but the Interface will remain open, allowing the user to continue the analysis of the code.

In order to explain how to use the tool with more details a simple example code will be taken, the steps of its modification and of the utilisation of the tool will also be explain. The chosen code is the message in a ring (see Appendix D The test codes page 75) for its simplicity and also because it can illustrate most of the functionalities. Obviously knowledge of C and MPI is required to understand this section. It is also assumed that the MPI Visualisation project (i.e. the Profiler library mpi_wrap and the Interface executable mpidisplay) are installed in ~/local and that the PATH environment is set up (meaning that header files are in ~/local/include, binaries in ~/local/bin and libraries in ~/local/lib and that the bin folder is included in the PATH).

\(^1\)Waiting for one to validate an MPI function before performing it
\(^2\)Synchronisation could be set up using either using command line argument or the MPIWrap_set_sync() function.
4.2.1 Code modification

The original code is a simple 60 lines long code, that performs communications from a processor to its left neighbour while receiving from its right neighbour. As the two communications cannot be synchronous (it would deadlock), the sending is non-blocking. However the request are not waited on (the code is commented). The original code is also using two arrays, one to send and one to receive. The code is available in the Listing C.6 (page 73) and the patch including all modification discussed here is displayed in the Listing C.7 (page 74). Compiling and running the original code is done using the normal commands

```bash
> mpicc ring.c -o ring
> mpiexec -n 4 ring
```

Two operations have to be performed to use the Profiler:

- add a `#include <mpi_wrap.h>` close for the library
- add includes and linking information while compiling

The first modification is trivial. In order to allow the code to be compiling without the `mpi_wrap` library, it is recommended to add a pre-compiler MACRO definition checking, to know if the library is used or not (see the Listing C.7: header modification). The reason is that if the user wants to recompile its code without the Profiler’s library, only the linking stage and this MACRO could be removed, without modification of the code (if the MACRO does not exist the user’s code will not include the library header and not use `mpi_wrap` functions).

The compilation and running processes do not change a lot. Note that the execution however will stop, as the Profiler part of each MPI process is waiting for the Interface to connect on it in order to continue.

```bash
> mpicc ring.c -o ring -DWITH_MPIWRAP -I~/local/include -L~/local/lib -lmpi_wrap
> mpiexec -n 4 ring
```

The above modifications are enough to make the code use the redefined MPI functions. However it is not sufficient to allow the usage of the command line options and some code has to be added to register the arrays.

It is very important to give the `argc` and `argv` arguments to `MPI_Init()`, rather than giving `NULL` (as it is in the Listing C.6: important line). Otherwise the `mpi_wrap` library does not have access to the command line arguments and uses its default settings. Thus the patch adds them to the `MPI_Init()` code (Listing C.7: `MPI_Init()` modification).

To register an array a call to `Array_register()` has to be explicitly performed from the user’s code. For the same reasons than previously such code will be wrap around
MACRO checking, insuring that it will not be included if the library is not used. Registering an array is ensured to work only if the array is linear in memory, that is ensured by the C standard for static arrays. In this example the arrays are statically defined, of size \( N \), their registration is straightforward (see Listing C.7: Array_register () modification).

Compilation and running are not affected by the latest modification, the code can now be used with all the Profiler's functionalities and the arrays displayed on the Interface. A simple addition of ten lines of code was enough to use the tool. However for the next step, using the Interface, it will be assumed that the code is run with:

```bash
> mpiexec -n 4 ring --port-in-file port.txt
```

### 4.2.2 Running the Interface

The Interface is provided as the mpidisplay executable. It has to be started after the profiler, for the obvious reasons that it needs the ports to be published (and written on either standard outputs or file). Running the interface can be done from another console, or using pipeline techniques:

```bash
> mpiexec -n 4 ring | mpidisplay
```

The starting dialogue (see previous Figure 3.4 page 33) should appear, allowing the user to select the port source, in this case opening the file `port.txt`. Once the connection is performed, the MPI programs execute themselves and the information starts to appear.

### Statistics

The number of calls to each MPI function should be of ITERATION (10) and the total of calls of \( N \times \text{ITERATION} \) (40). The Communication Information should reveal that in total 40 sends were done, 40 receives performed, while the number of generated requests are also 40 and one can see that none was waited for. The user can then add the wait operation, re-compile and execute the program again to see that the number of waited requests will now be 40 as well.

### Communications view

In the communications view the processors could be moved and the communications pattern should become obvious (see Figure 4.3 page 54). The registered array also displays which cells were accessed in reading and writing, showing in that case that Ssend buffer was read andRecv buffer written on. The processors box could be moved in the view by the user, in order to allow reordering according to the type of communications.
Figure 4.3: The message in a ring communications view (8 processes).
Exporting information

Exporting information is done straightforwardly from the File menu, that only has this option. The user select a file where the XML data is written in.

Synchronisation with the user

In the case of that example a more interesting feature is the synchronisation with the user. In order to do that the user code has to be started using:

```plaintext
> mpiexec -n 4 ring --port-in-file port.txt --sync ISSEND
```

Or the synchronisation could be set up directly in the code using the MPI_Wrap_set_sync() functions at key points where synchronisation has to be turned on and off.

Now, as soon as the Interface is started the programs blocks, as any call to MPI_Isend() will require the user to confirm it. Each time the user confirms a call, a small ball will traverse the link from the sender to the receiver. Note that this animation is not linked to the actual MPI calls, it is just to symbolise that the communication will now be performed. If the user activates the auto continue option one can see the operation occurring in sequence, the ball will move from one process to another each time (Figure 4.3 page 54).

This can be used to show that MPI processes may be executed simultaneously but be at different point in the program, as the statistics may display a different number of sends and receives per processor. Moreover this synchronised view allows the user to see what data is used for the occurring communications, as their colour is different in the registered arrays (red and blue are active reading and writing, while yellow and grey are used to show past access).

4.3 Applications results

Some application are already presented previously. It is the case of the traffic model and the matrix transposition and of the message in a ring. More test cases will be develop in that section and the focus on resources utilisation will be performed. Run time is not computed, but the display of the information will be discussed if it is relevant; it is the memory impact that will be mostly investigated. In order to do so the size and top tools were used.

size is a Unix tool providing the size of an executable (total) and of several of its components. The text is the amount of text data in the executable. While the data is the amount of memory for static variables, the bss is size information used by the operating system to allocate the heap variables (dynamically allocated). The size tool provides the size in bytes (B).

The other tool used is top, it provides run time information about CPU usage and memory consumption. The field used is virt, that is the amount of virtual memory.
needed for the executable. It includes all the needed memory for the program, but does not necessarily implies that all this memory is loaded in RAM. The memory usage is displayed in kilo bytes (kB).

This section’s data were generated with usually unique runs, as it aims to present different memory usages for the software rather than performing a benchmark of the tool. All the tests were performed on the Ness front-end, an EPCC cluster available for students, as the timing were irrelevant, the main memory consumption comes from the Interface, that is anyway run on the front-end.

4.3.1 Simple codes

Hello world

A hello world program was done, that only requests its ranks and the size of MPI_COMM_WORLD. The impact in memory usage of the mpi_wrap library is not big, as it is not by essence a huge library (see Table E.1 page 78). The goal is to show the minimum memory consumption for both the Profiler and the Interface. The former is relatively small, adding around 1 kB during the execution on the 45 kB used for the original code. The Interface on the other hand is consuming 250 MB at least to run and in that case it is an empty program with no array displayed at all.

Message in a Ring: multi communicators

The mulcom code is a stress test of the Interface and a memory leak verification for the Profiler. The program generates 300 copy of MPI_COMM_WORLD, but the Profiler only stores 100 (each time 100 communicators are registered, the user’s code is unregistered them). This is a quite violent test, as usually the number of communicators is less than the number of processors. During all the execution of this test the Interface was very slow to start, as the MPI code executed in few seconds (between 1 and 3 for 2 to 16 processes) and it had to cope with the generation of all the objects at once. The memory consumption is not that bad, as not arrays are displayed, as 540 MB are used for 16 processors (Table E.2 page 79).

Traffic: array effect

The traffic model code was used to investigate the effect of the size of registered arrays on the interface. For instance the size of each process’ array is the same, therefore the more processes the more memory will be used. Three size of arrays were used: 32, 8, 192 and 32, 768 elements. While the first size is relatively small, the two others are closer to real use of medium-sized programs (32k elements per process is not that big as it will be shown in the MPP coursework section). The memory consumption of the Profiler is not very big, as the size of the array does not affect the registered information. On the other hand, the Interface consumption is quite important, going up to 3 GB of memory for 16 processes with the biggest array size (Figure 4.4 page 58). For this case the total of array cells in the Interface is 524, 288. Moreover each cell is divided in two
Table 4.1: Array size for each process and their overall total and halo size.

<table>
<thead>
<tr>
<th>Decomposition</th>
<th>per process</th>
<th>total</th>
<th>halo</th>
</tr>
</thead>
<tbody>
<tr>
<td>2x1</td>
<td>50,052</td>
<td>100,000</td>
<td>1,800</td>
</tr>
<tr>
<td>2x2</td>
<td>25,220</td>
<td>100,880</td>
<td>2,576</td>
</tr>
<tr>
<td>4x1</td>
<td>25,284</td>
<td>101,136</td>
<td>2,832</td>
</tr>
<tr>
<td>2x4</td>
<td>12,804</td>
<td>102,432</td>
<td>4,128</td>
</tr>
<tr>
<td>8x1</td>
<td>12,900</td>
<td>103,200</td>
<td>4,896</td>
</tr>
<tr>
<td>4x4</td>
<td>6,468</td>
<td>103,488</td>
<td>5,184</td>
</tr>
<tr>
<td>16x1</td>
<td>6,708</td>
<td>107,328</td>
<td>9,024</td>
</tr>
</tbody>
</table>

triangles, that results in more than a million elements displayed in total, that is slightly less than 3 kB of memory used per triangle.

4.3.2 The MPP coursework

The MPP coursework is considered, for this project, as the maximum size of application that could run through the tool (at this stage of the project such a program is considered huge). The four available versions are the *MPP casestudy, MPPcwk thomas, MPPcwk paul* and *MPPcwk lesleis*. The details, about the code’s authors and information are explained in the Appendix D *The test codes* (page 75).

The image used is 384 by 256 pixels (98, 304 pixels) and all others relevant details will be given for each test.

**MPP casestudy**

The casestudy code uses only one communicator and splits the array. Only one array is registered in the profiler, the total number of cells is roughly 100,000 elements (Table 4.1). The memory usage however is important but not especially important, 1.5 GB are used maximum and the display time is reasonably short.
Figure 4.4: The mpidisplay memory consumption for different traffic model’s size.

Figure 4.5: The mpidisplay memory usage for different implementation of the MPP coursework.
MPPcwk thomas

This code is certainly the less evolved. The processes do not split the array, all of them hold the equivalent of the whole image. Therefore it is an interesting code to test the memory consumption, so far it is the code that requires the most memory. It uses two communicators and registers only one array. The used decompositions are 2x1, 2x2 and 2x4, but the amount of memory displayed will stay the same: 99,587 cells per process (a halo is added for computation safety).

Obviously it uses twice the amount of memory than any other code on two processes and so on. This memory requirement is so important that it could not have been run on 16 processes on the front end, requiring too much memory for both the Interface and the normal MPI code. Also when run the Interface is quite slow, as it receives a lot of messages and has to cope with the update of the communication view simultaneously.

MPPcwk paul

Paul’s version is interesting for several reasons. First of all it has a deep halo implementation, that is unusual for the coursework code. It also does not wait for half of the communications, showing in a real context that forgetting to wait is possible. Finally it is a more advanced implementation than thomas’ version, as it splits the data among the processes. It does it with static arrays, that need recompilation each time the input file changes. The memory consumption is relatively good for the Interface, as the maximum usage is 1.5 GB for 16 processes (registering only the main array).

Using the synchronised mode and registering an extra arrays (the edge array is used during the deep halo swapping), allows one to see the steps of the deep halo swapping. The first one is exchanging data from the edge array, that is the temporary array used for computation and second is a bigger classical halo swap (see Figure 4.6 page 60).

MPPcwk lesleis

This version of the MPP coursework is important, as it provides a code that uses dynamic arrays. It uses the arralloc () tool provided during the MPP classes, that allocates dynamic arrays but allows them to be linear in memory to be used with the MPI routines. No more is needed from the usage of the array visualisation, this code allows a proper testing of such features. As it uses dynamic arrays, it does not need to be re-compiled to change the source file, but therefore needs command line arguments to be verified. This was another important point, in order to check that the Profiler part was not disturbed by other programs’ arguments.

As for the two other implementations that handle 2D decomposition, it uses a Cartesian communicator. In order to have different results than Paul’s version the master buff array was registered as well (it is a buffer than only the master process has, in order to collect results and write them on the disk). Therefore the total amount of cell registered is the double than in Paul’s or the casestudy versions. The Interface copes quite well with it, using 1.8 GB with 16 processes. The display time is relatively slow, as for the other codes.
Figure 4.6: The deep halo swapping steps.
Overall memory usage and display speed

On most of the big code running the Profiler does not slow down the MPI code significantly. However it was said before that the Interface is slow. The main reason is that it receives all the messages in a very short amount of time and has to deal with them resulting in sending a lot of signals. Some improvement could be done on that, allowing either the Interface to pack messages and only send a certain amount of messages per seconds, to allow enough time for the software to react. Or the Profiler could send messages in a window and wait for the Interface when the window is full. Obviously when running in synchronised mode the Interface is not slow at all, as the Profiler is blocked until the user validates communications.

The memory consumption is higher than expected. It is, nonetheless not critical, as using 2 GB of RAM for an application is possible on nowadays desktop machines. However, according to the amount of data stored and their type, it is a weakness of the Interface to use that much memory. The future development have to take that in account and change the memory need to a lower consumption.

4.4 Requirements compliance

Most of the functional requirements (see Appendix B Project Functional Requirements page 68) have been completed. The Data View is achieved, as array registration, graphical display and derived datatypes are implemented. However the communication profiling ones are not totally finished. Point to point communications are implemented for most of the functions, but not all the available ones. Synchronised communication is implemented, as well as the display of communication with “animations”. The generation of a log file is not completed, but the basis of its development is available in the sources (exporting to XML); it is the same for the communication time, that is not finished. In fact only the Global communications are not implemented at all.

The non-functional requirements are also mostly completed. The code modification needed to use the tool is as minimal as possible, but it is still hard to define the tool as stable, for the reason that it was not used by a lot of people and therefore deserves more real testings. The Interface was developed to be as easy to use as possible, it mostly achieving that goal, although few people tested it as well.

However the tool resources utilisation has to be improved. The memory consumption is far too high for the information it displays. Especially when knowing that the code used as test cases are not very complex. Moreover some developed functionalities could be improved, especially the missing MPI functions support, as even though the most used of the point to point communication are implemented some are missing (MPI_Rsend(), MPI_Envsend(), MPI_Waitany(), MPI_Waitsome() and MPI_Sendrecv() are not implemented).
Other functionalities

During the development several potential functionalities were found and were put aside as possible later development.

- On the subject of availability and ease of installation, the usage of GNU tools package to generate a configure script could be very interesting. It would allow the user to configure the MPI library linking (actually done manually) and to check that the needed library are available (Qt for instance).

- Providing more default positions for the processors in the communication view could also be interesting, as the current one is not very useful. Some graphs algorithms could even be used to automatically arrange the processors in a tree that limits the number of link intersections.

- Adding symbols on the links when a communication is waiting for the user to validate it could also be interesting. Highlighting the link when the mouse is over the request in the requests list could also help the user visualise the waiting communications.

- Grouping communications could also be interesting. Allowing the user to regroup waiting communication and define them as patterns, that will be automatically validated together. This could allow, for example, halo swapping to be considered as a unique action and displayed as such (all communications occurring at once in the Interface).

- Redefining the communicators’ or processors’ names in the Interface could also be interesting, especially for data export purposes.
Conclusion

The project ran for three months. It started from the Project Preparation that defined the requirements and created the technical basis of both the Profiler and the Interface. Almost all of its goals are now completed. The tool provides communication views of simple MPI codes, with memory display and user synchronisation. The result is a portable tool, on platforms where Qt and MPI are available, that are most of the current computers (if not all used in HPC). It was successfully deployed on two architectures, a shared memory cluster (Ness) using MPICH2 and a typical desktop machine, using OpenMPI. The code source was also compiled using different compilers, in order to refine its correctness.

However it is not yet a finished tool. Some functionalities are still missing, as the global communications implementation that is an important part of most of the MPI codes. Since the project was developed as a prototype, a proof of concept, it therefore also lack maturity. However the tool does not have users yet, but whenever it will complete its missing functionalities it could be considered as a proper tool to help learning MPI. On technical matters the software design of the Interface has to be improved. Mainly because its memory consumption is not acceptable for the actual aimed codes’ complexity. The Profiler part, closer to a final design, proposes almost all the functionalities of a complete tool. Moreover the tool has proved to be robust enough to cope with most of the testing cases.

Despite the above limitations, the tool was successfully tested on four medium sized MPI code. These codes were written earlier in the year, by different authors and have no link with this project. Moreover the tool aimed for simplicity of use and as little code modification as possible and achieved it. It also proved to be adaptable enough to cope with the different codes, that were using Point to Point communications, derived datatypes and several communicators. It was used to display the communications in real time and visualise the accessed memory during the sends and receives.

The MPI Visualisation project is a good start as a promising tool. Although it still has problems to address, it is functional and adaptable enough to be used on simple MPI codes. It is portable, open source and available to the public. If beginners - and teachers - find it interesting it could be a useful visualiser to help learning MPI programming.
Bibliography


[10] Pacheco PS. Parallel programming with MPI. Morgan Kaufmann; 1997.


Appendix A

Original Project Proposal

Real-time Visualisation of MPI programs  David Henty

One of the problems with MPI programming is that it is very difficult to debug incorrect programs. Tools like VAMPIR can display the communications patterns of MPI programs by producing a trace file during execution and enabling the user to view the file as a timeline afterwards. Unfortunately, this is only useful if the program runs to completion which is usually not the case when you have a bug! It would also be useful to track MPI communications at runtime for training and education purposes, allowing new users to see what their programs are doing, or to run standard examples and follow their execution so they can understand concepts such as synchronous/asynchronous modes and blocking/non-blocking operations.

The project is to develop a tool/library that, for each MPI processes, pops up a window that shows real-time information about its execution. For example, it could just say what routine was being called (“currently in MPI_Send”), give more details (“Calling MPI_Send to send 14 real numbers to rank 4”) or display the operations graphically (eg boxes showing all the pending sends and receives, animations showing messages matching up at runtime etc etc). This tool would then be run on a set of test programs from simple examples all the way to full applications to see how useful it is in practice. Possible extensions include halting execution until the user hits a button (“click here to continue”) which could be very useful in illustrating concepts such as collective communications: the routine will not complete until the user has clicked “go” for all MPI processes. Another possibility would be to display where in the source code each process is at any one time.

It is quite simple to do this in practice as the MPI library has a separate “profiling interface” that enables all MPI calls easily to be intercepted by the user. Here, we would then display information about the call in some way (eg write text to a window) before calling the real MPI routine.
The tool could easily be developed and tested on a single workstation with all MPI processes displaying information on the same screen. However, it would be more interesting to run on a real cluster like the EPCC training room machines. Here, a window would appear on each screen where an MPI process was running and there would be interactions between different machines in the room. A user at one screen might have to call to a user at another screen for them to initiate a receive operation so that the first user’s synchronous send can complete.

The tool should work with both C and Fortran, but will itself be developed in C. A good knowledge of C programming is therefore required. Previous experience in graphics programming would be useful but not essential.
Appendix B

Project Functional Requirements

These requirements were introduced in the Project Preparation report.

1. Communication profiling
   (1.1) *point to point*: communication from one processor to another.
   (1.2) *global*: use of general communication routines.
   (1.3) *using different communicators*: registering and selecting the communicators to profile.
   (1.4) *communication time*: display the time when communication occurred and the duration of the operation.
   (1.5) *step by step view*: providing a blocking synchronised view of the communication, showing step by step what is going on for each processor (require either 11.1 or 11.2)
   (1.6) *display communication with graphical “animation”*: display the occurring communication with simple animation (require 11.5).
   (1.7) *generate a log file*: either using standard log formats (Vampir’s for example) or a dedicated one.

2. Data view
   (2.1) *register an array*: see information regarding an array when used in communications by registering it to the profiler.
   (2.2) *display graphical view of registered data*: when used during communication display which part of the data are transferred (require 11.1 or 11.2 and 22.1).
   (2.3) *recognise derived data types*: in order to display the graphical view using simple data types first (vectors) or more complex ones later (enhance 22.2).
Appendix C

Code Listings

Example of the MPI profiling interface:
- C.1 Example: MPI profiling interface, header file.
- C.2 Example: MPI profiling interface, profiler file.
- C.3 Example: MPI profiling interface, user file.

Example of the Template Linked list
- C.4 Example of the template linked list usage: scalar.
- C.5 Example of the template linked list usage: array.

Example of MPI ring communication
- C.6 MPI example: Message in a ring.
- C.7 Patch to use mpi_wrap with the message in a ring (Listing C.6).
Listing C.1: Example: MPI profiling interface, header file

```c
#ifndef MPI_WRAP
#define MPI_WRAP

int MPI_Init (int *argc, char ***argv);
#endif
```

Listing C.2: Example: MPI profiling interface, profiler file

```c
#include "mpi_wrap.h"
#include <mpi.h>
#include <stdio.h>

int MPI_Init (int *argc, char ***argv)
{
    int ret;
    fprintf(stderr, "Prof:(MPI_Init(...))");
    ret = PMPI_Init(argc, argv);
    return ret;
}
```

Listing C.3: Example: MPI profiling interface, user file

```c
#include <mpi.h>
#include <stdio.h>
#include "mpi_wrap.h"

int main()
{
    int rank=0, pop=0;
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &pop);
    if (rank == 0) printf("I'm the master of %d puppets.
", rank, pop);
    MPI_Finalize();
    return 0;
}
```
/* int linked list */
int compareInt(void* elema, void* elemb)
{
    return ((int)elema) == ((int)elemb) ? 0 : 1;
}

void linkedlist_int()
{
    LinkedList_T* list = NULL;
    Cell_T* search = NULL;

    list = List_malloc(&compareInt, sizeof(int));
    List_pushBack(list, 1);
    List_pushBack(list, 2);
    List_pushBack(list, 3);

    search = List_search(list, 0);  // returns NULL
    search = List_search(list, 1);  // returns pointer to Cell_T
    printf("%d\n", (int)search->object);

    List_removeFirst(list, 2);     // returns SUCCESS
    List_removeFirst(list, 2);     // returns FAILURE
    List_free(list);               // always returns SUCCESS
}

Listing C.4: Example of the template linked list usage: scalar.
int compareArray (void* elema, void* elemb)
{
    int *a = (int*)elema; int *b = (int*)elemb;
    int i;

    for (i = 0; i < N_Elem; i++)
    {
        if (a[i] != b[i]) return 1;
    }

    return 0;
}

void linkedlist_array()
{
    int *a = (int*)malloc(sizeof(int)*N_Elem);
    int *b = (int*)malloc(sizeof(int)*N_Elem);

    LinkedList_T* list = NULL;
    Cell_T* search = NULL;

    list = List_malloc(&compareArray, N_Elem*sizeof(int));

    List_pushBack(list, a);
    search = List_search(list, b); // returns NULL

    List_pushBack(list, b);
    search = List_search(list, b); // returns pointer to Cell_T
    // Cell_T::object to access the object but needs static cast
    printf("%d\n", ((int*)search->object)[N_Elem-1]);

    List_removeFirst(list, a); // returns SUCCESS
    free(a); a = NULL; // remove does not free memory

    List_freeObject(list); // always returns SUCCESS
    b = NULL; // remove Object does free memory
}

Listing C.5: Example of the template linked list usage: array.
```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define ITERATIONS 10
#define N 10

int main(int argc, char* argv[]) {
  int mess1[N], mess2[N];
  MPI_Request leftReq;
  int run;
  int rank, size;
  int left, right;

double _time;

MPI_Init(NULL, NULL); // important line

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

left = (rank+1)%size;
right = (rank-1+size)%size;

_time = MPI_Wtime();

for (run = 0; run < ITERATIONS; run++)
{
  MPI_Isend(mess1, 1, MPI_INT, left, 0, MPI_COMM_WORLD, &leftReq);
  MPI_Recv(mess2, 1, MPI_INT, right, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    //MPI_Wait(&leftReq, MPI_STATUS_IGNORE);
}

MPI_Finalize();

fprintf(stderr, "Ring(%d): completed in %1fs \n", rank, MPI_Wtime() - _time);

return EXIT_SUCCESS;
}
```

Listing C.6: MPI example: Message in a ring
Listing C.7: Patch to use mpi_wrap with the message in a ring (Listing C.6)
Appendix D

The test codes

In order to test the Profiler and Interface several typical MPI codes were used, developed. This appendix will introduce them briefly.

Message in a ring

The message in a ring is a simple MPI program where each process sends data to one of its neighbours and receive therefore from the other (see Figure D.1). This program was developed to investigate the difference between several send and receive possibilities proposed by the MPI standard. This version will use a communication pattern of:

1. Asynchronous send (generates a request object)
2. Standard receive
3. Wait for the send’s request

The code doesn’t involve a lot of transfers, and can be modified to use a lot of different communicators (mulcom version). The code is available in the tests/rings folder.

Figure D.1: The ring communication pattern with 4 processors.
Traffic model

The traffic model simulation was a simple domain decomposition model, where a process is responsible for several cells of road. Each cell can be either occupied or empty; a car moves to an empty cell forward to it (see Figure D.2). Therefore some communication has to be made to send cars across the neighbour processors: check if the next cell is empty (a very simple 1D hallow swapping).

The code can be used to visualise simple arrays (1D) and halo swapping. It has an erroneous version that does not swap the correct cells for the halo. The sources are available in the tests/traffic folder. Compiling with the CORRECT MACRO will perform the correct halo swapping.

The Message Passing Programming case-study and coursework

The goal of the case-study was to develop an MPI version of a reverse edge-detector. The edge detection in an image generates a black and white file that contains average values. It is possible to reverse this process, but it is very computational intensive (to a certain extent) as the process is done in several steps. The algorithm uses a stencil averaging on the whole image and is therefore typical of scientific code usage.

The MPP case-study is the correction given by David Henty to the 1D decomposition problem, that was to split the image on one coordinate. It can be found in the tests/MPPcasestudy folder.

The MPP coursework was the same as the MPP case-study but using 2D decomposition (see Figure D.3 page 77), therefore having more complex halo swapping (the vector datatype becomes useful in this case). Several version of the code are available.

- **MPPcwk_thomas** is the author's version, which is the least evolved of the 3 proposed one. All MPI processes have a big array and they only do the computation in their allocated part.
- **MPPcwk_lesleis** was written by Lesleis Nagy, and uses dynamically allocated arrays. Each MPI process has only a smaller part of the image - the part it has to work on - and the master process will collect everyone parts to write the result.
• **MPPcwk_paul** was written by Paul Ross. It uses static arrays, as in the **MPPcwk_thomas** version. It therefore has to be recompiled for different images but also has a deeper halo option (can have several halos instead of the simple one).
Appendix E

Memory consumption

The memory consumption for different codes, that are not explicitly quoted in main graphs or tables.

<table>
<thead>
<tr>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>total</th>
<th>filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>462,630</td>
<td>399,624</td>
<td>152,696</td>
<td>1,014,950</td>
<td>mpi_hello_original.exe</td>
</tr>
<tr>
<td>719,744</td>
<td>504,792</td>
<td>152,968</td>
<td>1,377,504</td>
<td>mpi_hello.exe</td>
</tr>
</tbody>
</table>

(a) Size

<table>
<thead>
<tr>
<th>N proc</th>
<th>original</th>
<th>before interface</th>
<th>after interface</th>
<th>interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>45,592k</td>
<td>45,940k</td>
<td>45,944k</td>
<td>253m</td>
</tr>
<tr>
<td>2</td>
<td>45,592k</td>
<td>46,104k</td>
<td>46,108k</td>
<td>253m</td>
</tr>
<tr>
<td>4</td>
<td>45,592k</td>
<td>46,104k</td>
<td>46,108k</td>
<td>263m</td>
</tr>
<tr>
<td>8</td>
<td>45,592k</td>
<td>46,104k</td>
<td>46,108k</td>
<td>264m</td>
</tr>
<tr>
<td>16</td>
<td>45,592k</td>
<td>46,104k</td>
<td>46,108k</td>
<td>264m</td>
</tr>
</tbody>
</table>

(b) Runtime memory

Table E.1: Executable size and Memory (in Bytes) usage for the hello world code.
(a) Size

<table>
<thead>
<tr>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>total</th>
<th>filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>474,999</td>
<td>401,144</td>
<td>152,696</td>
<td>1,028,839</td>
<td>mulcom_original.exe</td>
</tr>
<tr>
<td>720,392</td>
<td>504,792</td>
<td>152,968</td>
<td>1,378,152</td>
<td>mulcom.exe</td>
</tr>
</tbody>
</table>

(b) Runtime memory

Table E.2: Executable size and Memory (in Bytes) usage for the mulcom code.

---

<table>
<thead>
<tr>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>total</th>
<th>filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>487,324</td>
<td>402,008</td>
<td>152,696</td>
<td>1,042,028</td>
<td>imagempi_original.exe</td>
</tr>
<tr>
<td>728,052</td>
<td>438,528</td>
<td>152,904</td>
<td>1,319,484</td>
<td>imagempi.exe</td>
</tr>
</tbody>
</table>

(b) Runtime memory

Table E.3: Executable size and Memory (in Bytes) usage for the MPP casestudy code.
Appendix F

Messages in strings or messages in a datagram

The choice between sending messages as formatted strings or inside a datagram was important for the project. This appendix details the size the messages will take in both cases. In order to do so an imaginary application is used. It only registers one communicator: MPI_COMM_WORLD and registers a 100x100 matrix call “forces”. It also uses a Vector datatype, of count 1, block length 1 and stride 100.

The Datagram needs at least 6 integers (needed for the send and receives, see Figure 2.4a page 17) and one block for names. For that example we can assume a maximum name of 20 characters, and a C struct with no padding.

The messages that are send will be:

- 1 initialisation message
- 1 communicator registration
- 1 array registration
- 1 vector datatype registration
- 16,000 Send, 16,000 Irecv and 16,000 Wait
- 1 finalise message

Size in a string  The size of each message is resumed in the Table F.1a (page 82).

The basic message is composed of message (2 char, 99 possible functions, 12 supported), time in (14 char, 10.3f is used), time out (14 char), sync (2 char, even if it is a cumulative value 2 char in decimal can support 10 options, 8 are available) and spaces (4 char) for a total of 36 characters. Each following message is composed of a basic message, the list of elements only concern their additional information, the total however is the sum of both basic and additional information.
• The Init message is composed of \textit{rank} (2 \texttt{char}, 99 processors...), size (2 \texttt{char}), space (1 \texttt{char}), for a total of 39 characters.

• The Register Communicator message is composed of \textit{id} (2 \texttt{char}), \textit{size} (2 \texttt{char}), \textit{name} (14 \texttt{char}), \textit{spaces} (2 \texttt{char}), for a total of 60 characters.

• The Register Array message is composed of \textit{id} (2 \texttt{char}), \textit{ndim} (1 \texttt{char}, up to 9 dimensions...), \textit{size} (3+3 \texttt{char} for the 100x100 matrix), \textit{name} (6 \texttt{char}), \textit{spaces} (4 \texttt{char}), for a total of 55 characters.

• The Register Datatype message is composed of \textit{id} (2 \texttt{char}, 13 basic datatypes, up to 99), \textit{type} (1 \texttt{char}, 2 types: basic or derived), \textit{count} (1 \texttt{char}), \textit{block} (1 \texttt{char}), \textit{stride} (3 \texttt{char}), \textit{spaces} (4 \texttt{char}), for a total of 49 characters.

• One Ssend or one Irecv message is composed of \textit{comm} (2 \texttt{char}), \textit{dest} (2 \texttt{char}), \textit{array id} (2 \texttt{char}), \textit{1st accessed} (5 \texttt{char}, 100x100 is 10,000 elements - no comma in C format), \textit{last accessed} (5 \texttt{char}), \textit{datatype} (3 \texttt{char}), \textit{spaces} (5 \texttt{char}), for a total of 60 characters. However there is 16,000 Ssend and 16,000 Irecv, pushing the total of each to 960,000 characters.

• One Wait message is composed of \textit{comm} (2 \texttt{char}), \textit{dest} (2 \texttt{char}), \textit{space} (1 \texttt{char}), for a total of 41 characters. As for the Ssend and Irecv, the 16,000 of them gives a final total of 656,000 characters.

The total data sent would therefore be of 2.5 MB.

\textbf{Size in a Datagram} \quad The size of the messages as used with a Datagram are resumed in the Table \textbf{F.1b} (page 82).

The problem of the fixed Datagram is from its nature: the same information has to be sent every time. In this case, and using the elements as listed before, the datagram needs 6 \texttt{int} (for the Ssend and Irecv messages) and a wait of storing the string of characters for the names. It is also important not to forget the basic message, composed of 2 \texttt{int} and 2 \texttt{double}. The problem that arises with a fixed Datagram is related to the wait \texttt{struct} are registered in MPI. Each field has to be registered, leaving therefore no room for dynamic size for the names. The sum presented assume either no use of names, or the use of names of maximum size 20 characters. The overall Datagram will therefore be either 64 Bytes or 84 Bytes (with 4 Bytes \texttt{int} and 8 Bytes \texttt{double}). Meaning that the overall communication will be (with the 3 \times 16,000 + 5 calls) 3.2 MB if the string field is always sent, \textit{versus} the 2.5 MB of the string-based communications.
<table>
<thead>
<tr>
<th>Message</th>
<th>Size</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>Init</td>
<td>5</td>
<td>39</td>
</tr>
<tr>
<td>Communicator</td>
<td>24</td>
<td>60</td>
</tr>
<tr>
<td>Array</td>
<td>19</td>
<td>55</td>
</tr>
<tr>
<td>Datatype</td>
<td>12</td>
<td>48</td>
</tr>
<tr>
<td>Ssend/ Irecv</td>
<td>24</td>
<td>60</td>
</tr>
<tr>
<td>Wait</td>
<td>5</td>
<td>41</td>
</tr>
<tr>
<td>Finalise</td>
<td>0</td>
<td>36</td>
</tr>
<tr>
<td><strong>Total sent</strong></td>
<td><strong>848,099</strong></td>
<td><strong>2,576,236</strong></td>
</tr>
</tbody>
</table>

(a) The size in a string (in characters or Bytes).

<table>
<thead>
<tr>
<th>Message</th>
<th>Size</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>All (no string)</td>
<td>24</td>
<td>48</td>
</tr>
<tr>
<td>All (always string)</td>
<td>44</td>
<td>68</td>
</tr>
<tr>
<td><strong>Total (no string)</strong></td>
<td><strong>1,152,120</strong></td>
<td><strong>1,204,240</strong></td>
</tr>
<tr>
<td><strong>Total (always string)</strong></td>
<td><strong>2,112,220</strong></td>
<td><strong>2,264,340</strong></td>
</tr>
</tbody>
</table>

(b) The size in a datagram (in Bytes).

Table F.1: Comparison of the size of message in a string or datagram.
Appendix G

Functionalities enhancement: examples

This appendix aims to show how to add a functionality to the Profiler and Interface by taking an example: adding MPI_Send() to the list of available MPI functions. But also adding support for an additional MPI Datatype: MPI_Type_contiguous(). The examples will be illustrated with pseudo patches for the code (this is the main reason why this appendix is quite long).

G.1 Adding support for MPI_Send

The Profiler

Adding a function overloading in the profiler is relatively easy.

```c
int MPI_Send (void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);
```

Listing G.1: Modifications to mpi_wrap.h

```c
+typedef enum SyncStatus { SYNC_NONE = 0,
+                     SYNC_SEND = 1,
+                     SYNC_SSEND = (SYNC_SEND*2)
+                     SYNC_ALL= (SYNC_SEND+SYNC_SSEND+
+                     SYNC_BSEND+SYNC_ISSEND+SYNC_RECV+SYNC_IRECV+SYNC_WAIT)
```

mpi_wrap.h The mpi_wrap.h header file is used to redefine the functions for the user. In order to add a new one, and make it available, the MPI prototype have to be added. If the function have to provide a synchronised with the user action (as it is the case for MPI_Send()) the enum SyncStatus has to be updated as well.
The `inter_comm` header holds the information shared by Profiler and Interface. In the case of a new overloaded MPI function it is its `Message` value (see following Listing).

```
typedef enum Message { MESSAGE_INIT, //= Initialisation of the MPI.
+ MESSAGE_Send,
+
Listing G.2: Modifications to `inter_comm.h`
```

This file holds the MPI overloaded function. The function’s core as to be added here and the proposed functionalities can be used as example on how to organise the implementation (Listing G.3 page 85).

If the function is available for user synchronisation the command line argument checking has to be added to the list. The `usage()` function has to be altered, in order to have the new option, as well as the `analyse_args_sync()`. If the function requires a new command line option, the option has to be added to the `usage()` function obviously, and its parameter looked for in `analyse_args()`. The following Listing shows the first method, as `MPI_Send()` simply needs a synchronisation addition.

```
// usage
− fprintf(f, "SYNC is composed of SEND, BSEND, ISEND, RECV, IRECV, WAIT or ALL with each value separated with a +.
+
Listing G.4: Modifications to `mpi_basic.c`
```

Listing G.4: Modifications to `mpi_basic.c`
int MPI_Send ( void * buf , int count , MPI_Datatype datatype , int dest ,
  int tag , MPI_Comm comm )
{
  int ret ;
  double time_in , time_out ;
  double time_f = MPI_Wtime () ;
  Register_Comm * commInfo = NULL ;
  Inter_message message [ INTER_MESSAGE_USER_SIZE ] ;
  commInfo = Comm_register_search ( comm ) ;
  if ( synchronisation & SYNC_SEND & & commInfo )
    {
      # ifdef DEBUG
      fprintf ( stderr , "! profiler (!%d) wants to sync a Send ...\n" ,
        world_rank ) ;
      # endif
      sprintf ( message , "%u,%d\0" , commInfo->id , dest ) ;
      check_memory ( buf , count , datatype , message ) ;
      if ( interface_waitUser ( MESSAGE_Send , message , & time_in ) != SUCCESS )
        {
          fprintf ( stderr , "! profiler (!%d) failed to communicate with
            the interface\n" , world_rank ) ;
          MPI_Abort ( internalComm , INTERFACE_ERR ) ;
        }
    }
  time_in = MPI_Wtime () ;
  ret = PMPI_Send ( buf , count , datatype , dest , tag , comm ) ;
  time_out = MPI_Wtime () ;
  if ( commInfo )
    {
      sprintf ( message , "%u,%d\0" , commInfo->id , dest ) ;
      check_memory ( buf , count , datatype , message ) ;
      if ( interface_send ( MESSAGE_Send , time_in , time_out ,
        SYNC_NONE , message ) != SUCCESS )
        {
          fprintf ( stderr , "! profiler (!%d) failed to communicate with
            the interface\n" , world_rank ) ;
          MPI_Abort ( internalComm , INTERFACE_ERR ) ;
        }
      time_out = MPI_Wtime () ;
      global_time += time_out - time_f ;
      return ret ;
    }
}

Listing G.3: Modifications to mpi_communication.c
The Interface

Now that the Profiler sends information about the function, the Interface has to cope with it. It requires the modification of only 3 files for simple MPI functions, but may require more modification if extra control has to be taken on the memory display for example.

MessageInfo class  The MessageInfo class is responsible for parsing the information from the Profiler. Therefore it contains the most modifications. First of all the name of the MPI function has to be added to the MPIFunctionName(). The status () function has to be altered as well, to cope with the status that will be displayed when the function is called. The fromString () function is the parsing method, it therefore has to be adapted to understand the information sent from the Profiler. Generally in the case of MPI_Send() the information are exactly the same as MPI_Ssend(). If the function should not be added to the statistics, the addToStats() functions has to be modified. In the other case, the readAction(), P2P(), \ lstinline{\ global}(), generatedRequest () and waitRequest () functions has to be modified according to the nature of the added functionality. The following Listing shows the modification done for MPI_Send().

```c++
// MPIFunctionName switch
+    case MESSAGE_Send:
+        return "MPI_Send";
+        break;
+    case MESSAGE_Ssend:

// status switch
switch (_message)
{
+    case MESSAGE_Send:
+    case MESSAGE_Ssend:

// fromString switch
+    case MESSAGE_Send:
+    case MESSAGE_Ssend:

// P2P switch
switch (_message)
{
+    case MESSAGE_Send:
+    case MESSAGE_Ssend:

Listing G.5: Modifications to messageinfo.cpp
enum CallInfo {
    N_Ssend = 0,
    +N_Send = 0,
    +N_Ssend,
}

enum TimeInfo {
    T_Ssend = 0,
    +T_Send = 0,
    +T_Ssend,
}

Listing G.6: Modifications to comminformation.hpp

// table creation: Call
+ tableWidget->setVerticalHeaderItem( N_Send, new
  QTableWidgetItem( parent->tr( "Number of Send" ) ) );

// table creation: Time
+ tableWidget->setVerticalHeaderItem( T_Send, new
  QTableWidgetItem( parent->tr( "Time in Send" ) ) );

// addInfo switch
+ case MESSAGE_Send:
+     C_item = tables[ Call ].item( N_Send, m.from() );
+     C_sum = tables[ Call ].item( N_Send, size );
+     T_item = tables[ Time ].item( T_Send, m.from() );
+     T_sum = tables[ Time ].item( T_Send, size );
+     break;

Listing G.7: Modifications to comminformation.cpp

CommInformation class  The class CommInformation stores information about communications occurring in a communicator. It is composed of 3 ‘arrays’: Call, Time and Meta, storing respectively number of calls to the function, timing in the function and the meta information about the functions in the communicator. The first thing to modify is to add the new function to the CallInfo and TimeInfo enums, that are the line number where the data are stored. The meta information should not be modified for the call of MPI_Send(), as all important information were added to the MessageInfo data-structure. But if the new functionality requires the addition of new meta knowledge the MetaInfo and generateMeta() function should be modified too. The new rows have to be added in the tables, and therefore the createTable method is going to have a few more lines. Then the addInfo() method have to me modified to cope with the new function. It is organised to have pointers to the call item (C_item - the processor’s function count), call sum (C_sum - the sum of calls to that function in the communicator), time item (T_item - the processor’s time in the function) and time sum (T_sum - the total time in the function in the communicator). Therefore new lines have to be added to the switch.
G.2 Adding support for new MPI Datatypes

The Profiler

mpi_wrap.h  As for the addition of any MPI function, this header has to be modified. For this simple case, adding the function definition is enough (MPI_Type_contiguous()), as shown in the following Listing.

```c
+int MPI_Type_contiguous (int count , MPI_Datatype old_type , MPI_Datatype *new_type_p);
```

Listing G.8: Modifications to mpi_wrap.h

inter_comm.h  The new datatype has to be understood when registering the datatypes, and therefore be added to the enum Datatype_type. The following listing shows how to do so.

```c
enum Datatype_type {
  BASIC_DATATYPE = 0 , //< Basic MPI C Datatype
  VECTOR_DATATYPE, //< A Derived datatype: Vector<count:blocklength:stride>
  CONTIGUOUS_DATATYPE //< A Derived datatype: Contiguous<count>
};
```

Listing G.9: Modification of inter_comm.h

datatype.c  Finally the main work is to add the datatype registration. The Vector datatype addition could be taken as an example, and the code is displayed in Listing G.10 (page 89).

The Interface

Handling new datatypes in the Interface is not very complex either. But it first requires the definition of a new class, that will represent this datatype.

Contiguous_Datatype class  This class is added to the datatypes.hpp, that holds the abstract class Datatype and its children Vector_Datatype. The class should hold information about the datatype and redefine the access() method, that returns the accessed element of an array given the first element and the number access. In the case of a contiguous datatype, it will be very simple. The modification are shown in Listing 10 and G.12 (both page 90).
Listing G.10: Modification to datatype.c


+**struct** Contiguous_Datatype: **public** Datatype
+
+ int count;
+
+ Contiguous_Datatype(unsigned int id, int count);
+
+ int* access(int first, int count, int* nAccessed) const;
+
+ virtual QString name(void) const;
+
+};

Listing G.11: Modification to datatype.hpp

+Contiguous_Datatype::Contiguous_Datatype(unsigned int id, int count): Datatype(id), count(count)
+
+ type = CONTIGUOUS_DATATYPE;
+
+}
+
+int* Contiguous_Datatype::access(int first, int d_count, int* nAccessed) const
+
+{
+ int *result = new int[d_count*this->count];
+ int k = 0;
+ +
+ for ( int i = 0; i < d_count; i++ )
+  +
+   for ( int j = 0; j < this->count ; j++ )
+  +
+    result[k] = first + (i*this->count)+j;
+    k++;
+  +
+ +
+  *nAccessed = k;
+  return result;
+}
+
+QString Contiguous_Datatype::name(void) const
+
+{
+  return
+  QString("Contiguous<count=")+QString::number(count)+QString(">");
+}

Listing G.12: Modification to datatype.cpp
MessageInfo class  As for any new message, its format has to be understood by the Interface. In the case of the Contiguous registration, the MESSAGE_REGISTER_DATATYPE already exists, and only its new datatype type has to be taken care of. In order to do so a new statement is added to the switch for both fromString() and generateDatatype() methods.

```cpp
// fromString, case MESSAGE_REGISTER_DATATYPE
+    case CONTIGUOUS_DATATYPE:
+        is >>= _size;
+        Log::println("Register CONTIGUOUS", (long)this);
+        break;

// generateDatatype
+    case CONTIGUOUS_DATATYPE:
+        neo = new Contiguous_Datatype(_datatype_id, _size);
+        break;
```

Listing G.13: Modification to messageinfo.cpp

DatatypeView class  This class is responsible to show the datatypes to the user, so one can have an idea of what is the effect of a datatype on a piece of memory. It is one of the latest functionality, and therefore is quite not refined. And adding another datatype is very good example of how it is not well engineered yet. The datatype’s information have to be added to an ArrayView object, that will display what elements are accessed. See the long Listing G.14 (up to page 92) (note that in the future the ArrayView object creation, and all its connect() calls, will be regrouped as a new class, as the same method is used for any datatype, and a very similar method is also used for memory display in the Processor class).
// newDatatype
+ else if ( d->type == CONTIGUOUS_DATATYPE )
+ {
+ Contiguous_Datatype* contiguous = (Contiguous_Datatype*)d;
+ ArrayView* neo = 0;
+ QLabel* neo_name = new QLabel(contiguous->name(), this);
+ QGraphicsScene* neo_scene = new QGraphicsScene();
+ QGraphicsView* neo_view = new QGraphicsView( neo_scene, this);
+ QToolButton* opt = new QToolButton(this);
+ opt->setText(tr("..."));
+ QMenu* arrayMenu = new QMenu(this);
+ QAction* display = new QAction(tr("Display datatype"), this);
+ QAction* dispMem = new QAction(tr("Display as Memory"), this);
+ QAction* dispMat = new QAction(tr("Display as Matrix"), this);
+ QPolygonF rect;
+ rect << QPointF(0, 0) << QPointF(ArrayView::cellWidth(), 0)
+ << QPolygonF(ArrayView::cellWidth(), ArrayView::cellHeigh()) <<
+ QPolygonF(0, ArrayView::cellHeigh());
+ int *size = new int[1];
+ size[0] = contiguous->count;
+ neo = newArrayView(1, size, true, rect, neo_scene);
+ delete [] size;
+ int acc;
+ size = contiguous->access(0, 1, &acc);
+ for (int i = 0; i < acc; i++)
+ {
+ neo->setElemState(size[i], CURRENTLY_USED);
+ }
+ delete [] size;
+ connect(display, SIGNAL(toggled(bool)), dispMem,
+ SLOT(setEnabled(bool)));
+ connect(display, SIGNAL(toggled(bool)), dispMat,
+ SLOT(setEnabled(bool)));
+ connect(dispMem, SIGNAL(triggered()), neo,
+ SLOT(displayAsMemory()));
+ connect(dispMat, SIGNAL(triggered()), neo,
+ SLOT(displayAsMatrix()));

Listing G.14: Modification to datatypeview.cpp (part 1/2).
connect(neo, SIGNAL(displayedAsMemory(bool)), dispMem, SLOT(setChecked(bool)));
connect(neo, SIGNAL(displayedAsMatrix(bool)), dispMat, SLOT(setChecked(bool)));
connect(display, SIGNAL(toggled(bool)), neo_view, SLOT(setVisible(bool)));
connect(opt, SIGNAL(clicked()), display, SLOT(toggle()));
gridLayout->addWidget(neo_name, gridLayout->rowCount(), 0);
gridLayout->addWidget(opt, gridLayout->rowCount() - 1, 1);
gridLayout->addWidget(neo_view, gridLayout->rowCount(), 0, 1, 2);
display->setChecked(true);
display->setChecked(true);
arrayMenu->setVisible(false);
dispMem->setChecked(true);
dispMem->setChecked(neo->memoryDisplayed());
arrayMenu->addAction(dispMem);
dispMat->setChecked(true);
dispMat->setChecked(!neo->memoryDisplayed());
arrayMenu->addAction(dispMat);
opt->setMenu(arrayMenu);
opt->setPopupMode(QToolButton::MenuButtonPopup);
opt->setAutoRaise(true);
datatypetypes.push_back(d);
}

Listing G.15: Modification to datatypeview.cpp (part 2/2).