Visual Parallel Programming

Vanya Yaneva

August 22, 2013

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2013

Abstract

The aim of this dissertation was to demonstrate that using *visual* parallel programming languages can provide natural and intuitive representations of the parallel applications in order to assist programmers and make parallel programming more accessible.

The approach taken was to define a new visual parallel programming language, called Vidim, based on the partitioned global address space (PGAS) programming model. By visualising different aspects of the parallel program and providing graphical controls through which the programmer to define them, Vidim aims to allow inexperienced parallel programmers to quickly develop working parallel code.

An end-to-end system was developed to enable application development in Vidim. It provides a visual interface through which programmers can implement parallel applications in Vidim and a translation tool, which turns the program into an executable. As an intermediate representation, the system uses Mesham textual code, which allows it to use the existing Mesham compiler to produce the executable.

Two parallel case studies, Prime Factorisation and Mandelbrot Set, were implemented using the Vidim system and compared against their implementations in other parallel programming frameworks. It was demonstrated that Vidim's visualisations provided simple and clear mechanisms for developing the parallel aspects of the cases studies.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank my supervisor, Dr. Nick Brown, for allowing to me work on this exciting topic and providing me with all the freedom and support I needed to pursue my own design. His guidance has been invaluable and his enthusiasm has been infectious.

I would also like to thank him and Iain Bethune, who co-supervised the project, for providing the necessary practical guidance and feedback, which continuously helped me stay within the scope of the dissertation.

Chapter 1

Introduction

The history of parallel computing can be traced back to the mid-1950s, when IBM starts designing the first supercomputers, as briefly outlined in [Wilson, 2013]. Since then, the advancement of parallel computing has enabled the development of extremely powerful, massively parallel machines which have allowed the rapid advancement of many areas of science. Currently, a look on the Top 500 Supercomputers website [TOP500, 2013] reveals that machines with millions of processors exist. On an everyday scale, parallelism quickly appears in the devices which people use all the time. Laptops have long been equipped with multi-core processors and even mobile phones now have quad-core processors.

In order to be able to take advantage of the computational power available through this parallelism, one needs to learn how to develop parallel applications. However, parallel programming has a set of challenges specific to it, which make it complex even for experienced software developers and as a result it is still way less common than sequential programming.

This inherent difficulty in parallel programming is the problem on which this dissertation focuses.

1.1 Challenges of Parallel Programming

Particular challenges associated with parallel programming include

- distribution of tasks across multiple processes,
- partitioning and distribution of data,
- communication and synchronisation between processes,
- protection of accesses to shared data.

Attempts to relieve the programmer from the effects of these challenges have resulted in new programming models, new parallel languages, extension libraries to sequential languages and novel visualisation techniques. Nevertheless, to this day most widely used remain traditional, textual, low-level, architecture-based approaches, due to their maturity, flexibility and the excellent performance they achieve.

Unfortunately, there is usually an inverse relation between the performance achievable with a certain parallel programming framework and its programmability, which means that few people are able to take advantage of the parallelism available to them.

1.2 Proposed Solution

This dissertation assess the ways in which *visualisation* can be used to address the particular challenges of parallel programming. Specifically, it proposes a new *visual* parallel programming language, in which separate processes and local memories are represented visually on the screen. Parallelism is still explicitly defined by the programmer, who specifies the data and tasks to each process and local memory individually. In this way, the visual representation provided by the language allows the programmer to separately define all parallel aspects of the application, as they would appear on each process during execution. This simplifies the programmer's task by giving them a visual representation of all of the program's aspects.

The visual programming language shall be named Vidim.

Vidim's Name

When considering the difficulties of textual parallel programming, an obvious disparity between the inherent linearity of text and the need for representation of parallelism becomes apparent. One of the ideas behind the proposed solution is that visualisation can be used to add a dimension to the program representation to represent parallelism. Thus, the name of the proposed parallel programming language came from the merging of the words *visualisation* and *dimension*.

A further reason to choose this name is the fact that the word "vidim" means "visual" or "visible" or even "to see" in many Slavic languages, which makes it fitting to the ideas of the project.

1.3 Project Goals

The project's overall goal is to assess the extend to which the proposed visualisation approach is successful in providing a more natural representation for parallel programs which simplifies their development. To achieve this, it sets out to do the following:

- Identify particular desirable properties of the visual parallel programming language.
- Define the new language in detail.
- Implement a visual system which enables the development of parallel programs in it.
- Identify and implement case studies, which can be used for the evaluation of the language.

1.4 Dissertation Outline

The rest of the dissertation describes the steps undertaken to achieve the above goals in the same order. In particular, Chapter 2 reviews some of the traditional approaches to parallel programming, as well as some of the existing visualisation solutions and uses a discussion of their advantages and shortcomings to draw a set of evaluation criteria, which are later used for the evaluation of Vidim.

Chapter 3 provides the conceptual definition of Vidim, while Chapter 4 describes the implementation of the visual system that enables parallel programming in it.

In Chapter 5, two case studies are identified and their implementations in Vidim are presented, discussed and compared against the evaluation criteria identified in Chapter 2. This allows for assessment of the ease of programming provided by the visual language.

Finally, Chapter 6 summarises the work carried out during the project, suggests possible avenues for future work and briefly analyses the working process, planning and risk management that were implemented in the course of the project.

Chapter 2

Background and Literature Review

The problem addressed in this dissertation and the proposed solution fall within two seemingly separate topics in programming language research. These are *parallel programming languages* and *visual programming languages*. As the aim of the dissertation is to demonstrate that the advantages of visualisation can be used to address the challenges of parallel programming, an overview of the existing work in both areas is needed.

The current chapter presents some of the traditional approaches to parallel programming together with their advantages and shortcomings. In addition, it discusses some of the arguments in favour of visualisation and reviews some of the work that has already been carried out in the area of *visual parallel* languages. The provided background is used to derive a list of desirable properties (evaluation criteria) for parallel programming languages, which is later used to set the context for the implemented solution and to evaluate it.

2.1 Parallel Programming

In [Browne,1994] the authors identify three conventional approaches to parallel programming languages.

1. Extend sequential languages with architecture-specific procedural primitives.
2. Develop compilers to automatically detect implicit parallelism in sequential programs.
3. Extend sequential languages to allow data partitions to be specified.

Traditionally, the first, architecture-based, approach has been most widely used. This is explained with the fact that the high performance which parallel programming targets is more easily achieved at the lower, closer to hardware level of programming. This means that parallel programming usually requires the developer to be familiar, up to

some level of abstraction, with the underlying architecture. There are two main parallel programming models, which are based on the two multicore memory architecture designs, namely

- **shared memory programming model**, which as the name suggests is used for the programming of shred memory parallel machines.
- **message passing programming model**, which is used for the development of applications for distributed memory architectures.

These two programming models exhibit different desirable properties. While programs developed with the message passing programming model display high levels of performance and flexibility, due to its low-level routines, developing parallel applications in the shared memory programming model is easier.

Therefore, another parallel programming model, which aims to combine the two, has been proposed by Stitt in [Stitt, 2010]. This is the

- **partitioned global address space (PGAS) programming model**, in which memory and data are distributed, but can be accessed directly from remote processes via a partitioned *global* address space, as the name suggests.

The two most widely used parallel programming frameworks for the shared memory and message passing models are OpenMP [Bull, 2012] and MPI [Henty, 2012] respectively. A textual parallel programming language which uses the PGAS programming model is Mesham [Brown, 2013]. The following subsections provide an overview of each of them, together with a discussion on their advantages and shortcomings.

2.1.1 OpenMP

OpenMP is a parallel programming framework designed for the programming of shared memory architectures. It is implemented as a set of compiler directives, library routines and environment variables, which extend traditional sequential programming languages, such as C, Fortran and C++.

In OpenMP, the programmer specifies which portions of the code can be executed in parallel by using compiler directives to explicitly define *parallel regions*. Threads execute the code in the parallel regions concurrently. The shared memory model uses a **single memory address space**, which allows each thread to access all data directly in the same way. In the parallel regions, the programmer needs to define what data is shared between the processes and what data is private. To avoid nondeterminism and ensure the correct execution of the program every time, the programmer also needs to explicitly synchronise all accesses to shared data.

With respect to ease of programming, OpenMP has the following advantages and shortcomings:

Advantages

- Data sharing between threads does not require explicit message passing, as each thread can access data directly using the single address space, which simplifies the process of programming.

Shortcomings

- It is the programmer's responsibility to ensure that accesses to shared data happen in the correct order. This adds complexity for the programmer not only because they need to think about accesses to shared variables, but also because of the textual mechanisms provided by OpenMP.
- In particular, synchronisation needs to be explicitly specified by the programmer with the use of compiler directives. In textual code, this requires putting the directives in their correct places, naming critical regions correctly and considering which operations need to be synchronised, which is often difficult and error prone.

2.1.2 MPI

MPI, which stands for Message Passing Interface, is a set of libraries, developed to enable the programming of distributed memory architectures using the message passing programming model. It provides a set of low-level routines, which can be used in the development of parallel applications in the C and Fortran programming languages.

In MPI, each process executes *the same* program on its own set of data. Processes have access only to their own data. They communicate with each other by sending and receiving messages, which are explicitly specified by the programmer. In addition, MPI provides a wide number of additional routines, which allow the programmer to perform tasks such as

- define communicators, which allow only a subset of processes to be used for particular tasks,
- organise processes in a Cartesian plane,
- use collective communications in which all process in a specified communicator participate

and many more.

From programmability point of view, MPI has the following advantages and shortcomings:

Advantages

- The low-level nature and high variability of its routines provide a lot of flexibility for the implementation of sophisticated high-performing applications.

Shortcomings

- Taking advantage of the above requires a high level of familiarity with the framework, which only comes with a lot of experience. For example, explicit message passing requires the programmer to consider many important factors, such as:
  - the type and size of the communicated data
  - which processes participate in the communication
  - should the communication be blocking or not
  - how should sends and receives be matched.

  Not only do these considerations complicate the programming of communications, but incorrect decisions on any of them lead to incorrect results and/or deadlocks.

2.1.3 Mesham

Mesham is textual parallel programming language, which uses the PGAS programming model. Through PGAS's global address space, Mesham allows processes to access remote data as their own. PGAS also supports distributed data structures, which makes data allocation and distribution in Mesham simple and flexible.

In addition, Mesham follows the *type oriented* programming model, in which parallel constructs and operations, such as variable allocation, data distribution and communication, are handled by *types*. The type oriented programming paradigm is introduced in [Brown, 2010]. In this way, a higher level of abstraction which makes programming more accessible is provided, while the expressiveness and flexibility of the language is retained through the multitude of types, which the programmer can choose to use. The full set of Mesham's type libraries, as well as further details on its concepts and tutorials, can be found in [Brown, 2013].

Mesham has the following advantages and shortcomings with respect to programmability:

Advantages

- Direct accesses to remote data eliminate the need to code explicit low-level communication details.
- Data allocation and distribution, as well as communications, are made easier to program both by the structures provided by the PGAS programming model and the abstraction provided by Mesham's types.

Shortcomings

- The need to position Mesham's types and keywords in their correct position in the linear textual code makes programming less intuitive and more error prone. This is one aspect of parallel programming in general, which can be greatly improved with the use of visualisation.

2.2 Visual Parallel Languages

Like almost all widely used programming languages, parallel languages have traditionally been textual. Nevertheless, a large amount of research exists in the area of visual programming languages. This is explained with the fact that, as Green and Petre argue in [Green, 1996], visualisation provides a representation of the problem and its solution that is closer to the programmer's mental representation than pure textual code.

Applying visualisation to parallel programming in particular is not a novel idea and aims precisely to address the problem of the higher level of complexity associated with parallel programming. A body of research exists in the area and a number of graphical parallel languages have appeared as a result. What they all have in common is representing the parallel program as graphs in which nodes and arcs correspond to different aspects of it. In this way they achieve two things, outlined in [Lee, 2003]. Firstly, they allow the programmer to specify which portions of the program can be executed concurrently by laying them next to one another, which adds another dimension to the program representation to account for parallelism. Secondly, they provide additional graphical artefacts as an intuitive representation of the programming aspects specific to parallel programming, such as concurrency, synchronisation and communication.

The existing visual parallel programming languages can be grouped in two categories based on the aspects of the program which their graphs represent. These are

- **data and control flow languages**, in which graph nodes represent sequential computations and the arcs represent dependencies between them.
- **communication model languages**, in which nodes represent processes, while arcs represent message channels between them.

Most of the research on visual parallel programming has focused on the first approach, resulting in languages such as CODE [Newton, 1992], HeNCE [Beguelin, 1991] and HiPPO [Lee, 2004]. The reason behind this is the fact that these languages not only provide the aid of visualisation, but also follow novel parallel programming models which reveal implicit parallelism. In contrast, the communication model approach has led to the development of one language, called VPE [Newton, 1994], which is based on the traditional message passing programming model.

A more detailed overview and discussion on both types of visual parallel programming languages is presented in the following two subsections.

2.2.1 CODE and HeNCE

CODE [Newton, 1992] and HeNCE [Beguelin, 1991] are two relatively mature visual parallel languages which are the result of years of research in graphical parallel programming. In general, both languages use graphs to represent the parallel applications in a similar way - nodes represent sequential computations, while arcs represent dependencies between them. The programmer architects a high level view of the parallel application by specifying the tasks and drawing the dependencies. Lower-level considerations, such as mapping processes to tasks, are handled by the language implicitly.

Figure 2.1 shows example codes in the two visual languages.

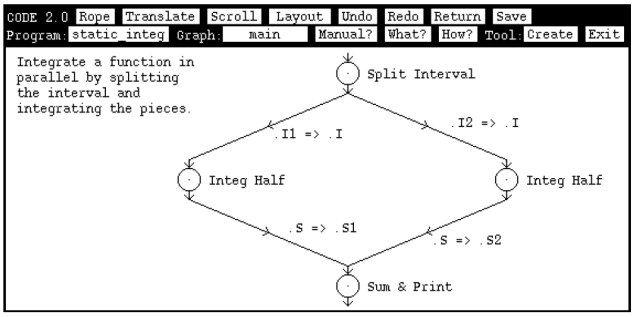
Below is a list of the visual languages' advantages and shortcomings with respect to programmability.

Advantages

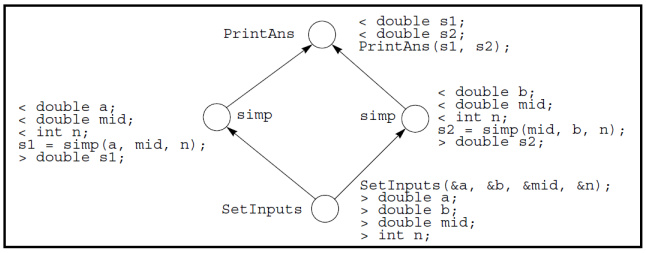
- The programming models used by the languages are both general and high-level, which relieves the programmer from needing to consider low-level architectural aspects of execution.
- Implicit parallelism is revealed, as the programmer only needs to specify dependencies between the sequential executions in the form of graph arcs.

Shortcomings

- While the general programming model simplifies the programmer's task by revealing implicit parallelism, it is less expressive and can be restrictive in cases when the programmer might need to specify parallel tasks themselves.
- Representing the program as a graph with nodes that correspond to sequential computations could lead to large and complicated graphs in cases of fine-grain parallelism.



(a) CODE



(b) HeNCE

Figure 2.1: Examples of CODE and HeNCE and HiPPO codes. Sources: [Newton, 1992] and [Beguelin, 1991] respectively

2.2.2 VPE

VPE [Newton, 1994] is different to the other visual parallel languages in that it uses the message passing programming model. Instead of sequential operations, nodes in VPE specify processes and instead of dependencies, arcs specify channels for communication between the processes. Figure 2.2 shows an example of a VPE graph. In that way, VPE provides a visual interface to the message passing model. The programmer still needs to specify explicitly the distribution and exchange of data between processes, but has the aid of the visual representations provided by VPE.

As with the other parallel solutions presented in this chapter, a separate list with VPE's advantages and shortcomings with respect to programmability is provided.

Advantages

- Graphical representations make the explicit definition of parallelism clearer.

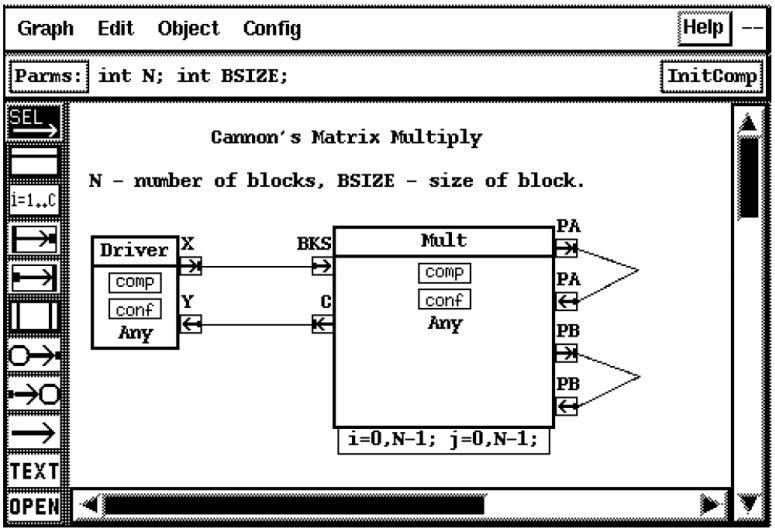


Figure 2.2: Matrix multiplication implemented with VPE. Source: [Newton, 1994]

Shortcomings

- The programmer needs to follow the message passing programming model and explicitly define the exchange of data between processes in the form of messages.

2.3 Evaluation Criteria

As pointed out in Chapter 1, it is widely accepted that parallel programming is more complex than sequential programming. The overview of existing parallel languages presented in the current chapter allows for the re-thinking of the challenges of parallel programming. As a result, a set of desirable properties and evaluation criteria for such languages is derived and used as a basis for the proposed solution and its evaluation. A new parallel programming language should:

1. **Have general architecture-independent programming model.**
   This will omit the requirement for the programmer to be familiar with low-level architecture details and make the language more accessible.

2. **Allow the abstraction of low-level communication details.**
   Explicit communication between processes is often difficult to program as it requires keeping track of what data is accessible to each process at all times. Thus, a parallel programming language should provide a way for the programmer to specify transparent access to data from all processes.

3. **Enable clear synchronisation between processes.**
   Explicit synchronisation points are critical for the correctness of parallel programs. However, ensuring that they are placed in the correct positions in the

program is difficult and error-prone. Therefore, a parallel programming language should make the specification of synchronisation points clear and unambiguous.

4. **Enable an simple and easy way to distribute data between processes.**
   Data distribution is an important aspect of any parallel program and a parallel programming language should provide a simple and easy way for the programmer to specify the allocation of data among processes.

Since the focus of this project is the *ease* of developing parallel systems, i.e. the programmability of the parallel languages, all of the above criteria focus on programability. However, performance is a critical aspect of any parallel programming language, as the purpose of parallel programming in the first place is acheiving better performance. Thus, in addition to providing more natural programmability features, any new parallel programming language should

5. **Demonstrate comparable application performance to other existing solutions.**
   Since there is often a trade-off between simplicity of programming and performance achieved, a slight degradation in performance might still be acceptable in cases when programmability is substantially improved.

2.4 Summary

The current chapter presented three widely used parallel programming models and some popular textual frameworks and languages based on them. It also reviewed some of the existing approaches to using visualisation to make parallel programming more accessible. In doing so, it identified the advantages and shortcoming of all existing approaches and devised a set of evaluation criteria for new parallel programming languages.

Table 2.1 shows the way in which the existing approaches compare against the programmability evaluation criteria. While it is not surprising that OpenMP satisfies more of them than MPI, it is striking that Mesham satisfies almost all criteria fully and the rest of them partially. This can be explained with the generality of the PGAS programming model and the abstractions provided by Mesham's types. In contrast, the visual parallel programming languages, CODE and VPE, satisfy very few of the criteria, which reveals the need for a new approach to visual parallel programming.

Evaluation Criteria	**OpenMP**	**MPI**	**CODE**	**VPE**	**Mesham**
General programming model	✓		✓		✓
Abstraction of low-level communication details	✓				✓
Clear synchronisation		partial	✓	partial	partial
Clear data distribution	partial				partial

Table 2.1: Comparison of existing parallel programming approaches with respect to usability evaluation criteria.

Chapter 3

Language Definition

The current chapter describes the design decisions behind the new visual parallel programming language, called Vidim, implemented in this project. It starts by briefly presenting the parallel programming model chosen for the language, namely PGAS. It continues with a description of the high-level graphical representation of the parallel program provided by Vidim by presenting the initial design idea first and describing the way this idea evolved into the final design afterwards. The chapter then individually describes the way various parallel and sequential program constructs are represented and used in Vidim.

Throughout the chapter, screenshots from the implemented system are used to provide illustration of Vidim's description. However, this chapter focuses entirely on the conceptual definition of the language. Details on the implementation are provided in the next chapter, Chapter 4.

3.1 Parallel Programming Model

The parallel programming model chosen for Vidim is partitioned global address space (PGAS). As discussed in Chapter 2, PGAS is a general parallel programming model, which uses a global memory address space through which all processes can access any data via standard reads and writes, but without assuming physical shared memory. In this way, it provides a general architectural overview for the programmer, while relieving them from the need to code explicit message passing for process communication.

Figure 3.1 shows an example of a distributed memory architecture with 4 processes. The key aspect of PGAS is that it allows each process to access each variable as their own from programming point of view. For example, process P0 can read and write variables `b` and `c`, which are allocated to processes P1 and P2 respectively, directly, exactly in the same way as it would its own variables.

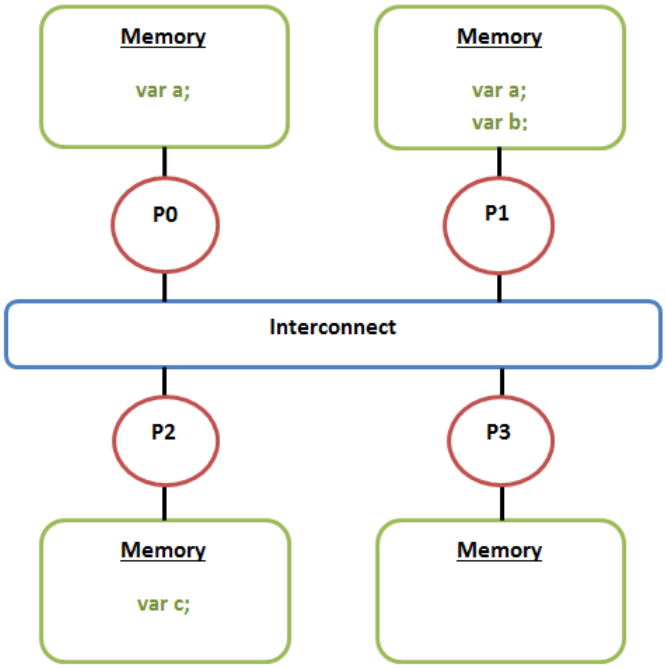


Figure 3.1: A distributed memory architecture with 4 processes.

While this programming model makes communications between processes easy to program, it should be noted that accesses to remote memories might incur performance cost, as they require the use of the shared interconnect. This might be an important consideration for the programmer from performance and scalability point of view, which is why replication of shared data across processes is commonly used with the PGAS programming model.

3.2 Evolution of Program Representation

The first and most important design decision for the new visual programming language was how to represent a parallel program using visual artefacts. While initially it was decided that the new visual language would use graphs as program representations, this idea changed in the course of the project work. In this section the original idea and the problems identified with it are reviewed and used to present and justify the final design.

3.2.1 Initial Design (as Presented in the Project Proposal)

The original idea for Vidim suggested that it should represent programs as graphs, much like the existing visual parallel languages reviewed in Chapter 2. The nodes would represent processes with associated local memories and the arcs would represent both

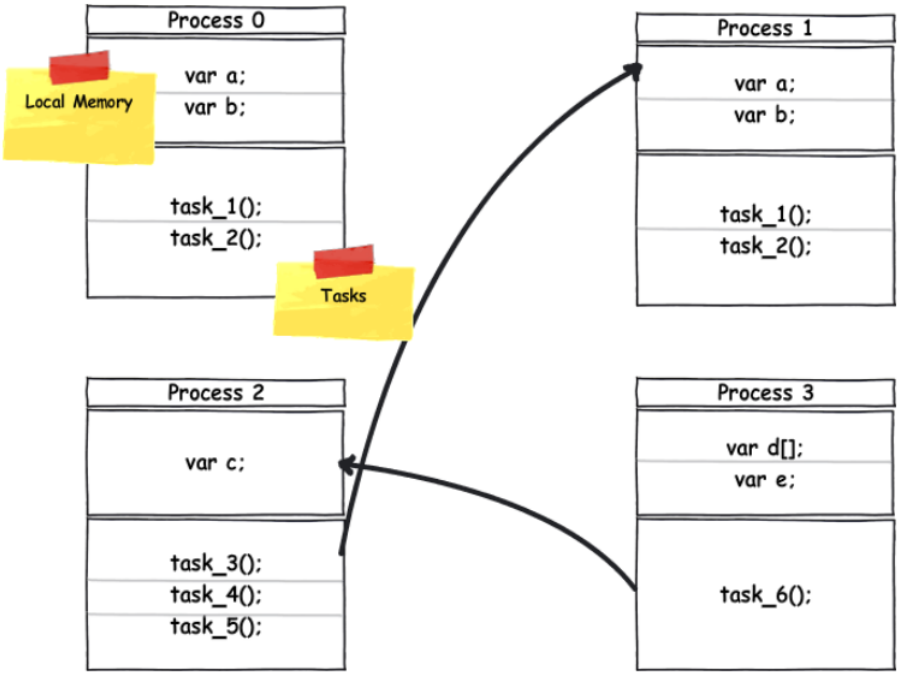


Figure 3.2: Mock up of the **initial** idea for a program representation.

accesses to remote memories and explicit synchronisations between tasks. Figure 3.2 shows a mock up of this initial idea.

However, a more detailed consideration of this type of representation identified the following problems with it:

- Allowing the user to add processes as graph nodes anywhere on the screen could quickly lead to large and complicated graphs which are hard to manage.
- Distributing the code across processes which are arbitrarily placed on the screen could make it additionally difficult to keep track of what tasks each process executes at what time.
- Using arcs to represent both communications and synchronisations between tasks is likely to add further complexity to the graph.

All of these problems relate to the fact that representing the processes as arbitrarily positioned graph nodes removes the ability for the programmer to easily specify and track the *order* of tasks performed in the program. Indeed, the code associated with each process would still be ordered, but the global order of tasks, the way it naturally appears in linear textual code, would not be represented. In this way, the initially proposed visual representation does not *add* a dimension to the program representation to account for parallelism, but *replaces* the one that represents the global order of tasks. It should be noted that while using arcs to represent explicit synchronisations between processes would allow the programmer to specify some global order when needed, that would not be a clear and intuitive way of doing it. It would require the programmer to carefully

position the two ends of the arc between the tasks described in two separate nodes of the graph.

What is more, using arcs to represent accesses to remote data was found redundant, as the PGAS programming model allows for these to be treated in the same way as local accesses.

Due to these considerations, the idea to represent the parallel program as a graph was changed.

3.2.2 Final Design

Figure 3.3 shows a mock up of the final visual program representation in Vidim. Processes are represented visually, together with their local memories and textual areas in which the programmer adds and modifies the textual code executed by each process. These areas are called *code areas*. Processes are displayed next to one another on the screen with their memory and code areas aligned. In this way, the user can write a program in the usual way, describing the tasks in order in the code areas, but write the code for each process in its respective area. The alignment of the code areas allows for the clear preservation of the global order of tasks, while the presence of an individual one for each process adds an additional dimension which represents parallelism.

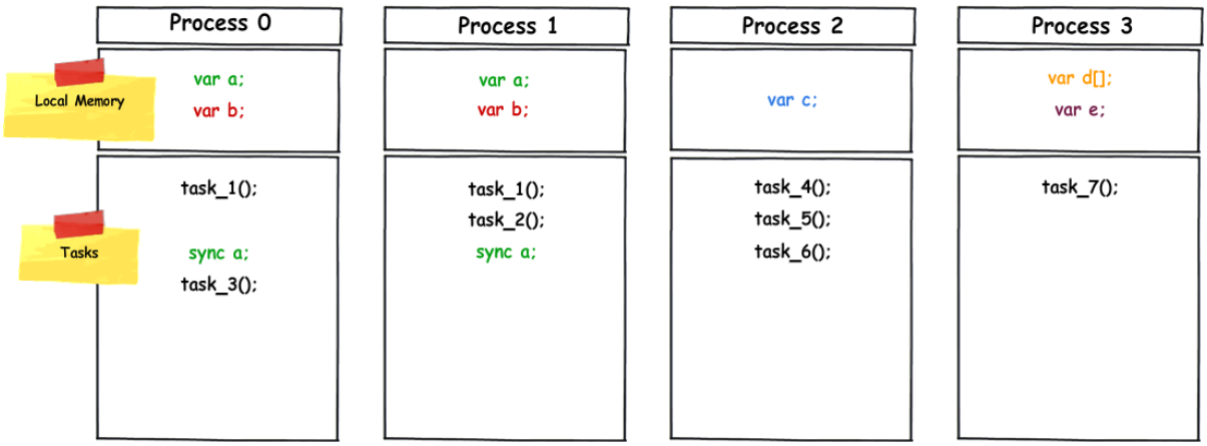


Figure 3.3: Mock up of the **final** idea for a program representation.

Furthermore, as shown in Figure 3.3, the local memory area for each process displays the variables allocated to it. In addition to this, each of the variables is colour coded, which makes recognising which processes they are allocated to easier. For example, in Figure 3.3 `var a` is allocated both to processes 0 and 1 and is coloured green in both memory areas. This separation of memory areas and visualisation of data allocation provides the programmer with a clear and intuitive way to specify and keep track of data distribution.

The colour coding of the variables, together with the aligning of the code areas, serve

an additional purpose in the final program representation. Explicit synchronisations on shared data updates, as well as collective communications in which all processes participate, and consequently wait for one another, are important aspects of parallel programming, as they are necessary for correctness. However, they impact performance and provide the risk of deadlocks when wrongly coded. Visualisation is particularly helpful in these cases. Aligning the synchronisations and collective communications in the code areas allows for a clear display of what tasks each process needs to complete before the synchronisation and what tasks come after it. For example, in Figure 3.3, both processes 0 and 1 complete `task_1()` and execute a synchronisation on `var a`, but process 1 also executes `task_2()` before the synchronisation. The visualisation clearly shows that process 0 needs to wait for process 1 to execute `task_2()` before it can synchronise and continue its own executions. In addition, colouring the code of the synchronisations and collective communications in the respective colours of the variables involved in them allows further differentiation of these important operations.

Finally, the initial design envisioned the use of drag-and-drop style controls to allow the programmer to position processes on the screen and draw arcs between them. Since this design changed into a much more structured one, in which visual components have strict positions, the drag-and-drop idea was replaced with that of using panels to provide the necessary graphical controls for visual programming.

These, as well as other particular aspects of using Vidim for parallel programming, are described in more detail in the two subsequent subsections. As their titles suggest, the first one describes the language's parallel constructs and the second one - its sequential aspects.

3.3 Parallel Constructs

In addition to the main program representation presented in Figure 3.3, the visual language provides an area on *the left-hand side* of the screen, which allows the user to specify some parallel aspects of the program. Figure 3.4 shows this area when implemented.

This section discusses each of the parallel constructs provided by Vidim in detail.

3.3.1 Number of Processes

In Vidim, the number of processes, on which the program is executed, is specified at compile time. As shown in Figure 3.4, the programmer enters the number or processes in a designated field. As a result, the specified number of processes, together with their memory and code areas, automatically appear on the screen. The programmer can then allocate variables to their memories and add tasks to be executed by them.

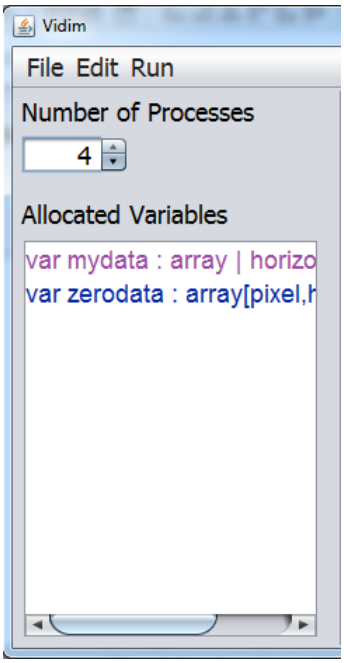


Figure 3.4: Vidim's parallel constructs' area.

Other aspects of the visual parallel language automatically take into account the entered number of processes as well. For example, the visual panels which enable the programmer to allocate variables to specific processes only allow allocation to processes which exist. For example, the programmer is not allowed to allocate a variable to process 10 when they have specified that the program will execute on 8 processes. In textual programming languages, errors like this would be detected during compilation if at all, while Vidim is capable of helping the programmer avoid them altogether.

3.3.2 Allocated Data

Variables in Vidim are declared and allocated via a right-click in the "Allocated Variables" area on the left hand side of the screen. Once a variable is declared, the programmer can specify different aspects of the variable's usage in a separate panel, called the *type panel*, which appears when they right-click on the variable. Figure 3.5 shows the type panel after it was implemented.

As seen in the figure, via using the controls in the type panel, the programmer can specify the following:

- The variable's name.
- The variable's *element type*, which can be *Boolean*, *Integer*, *Double*, *Character*, *String*, *Float*, *Short*, *Long*, *File* and *pixel*
- whether the variable is an array and the sizes of its dimensions, which can be either numbers or the names of number variables.

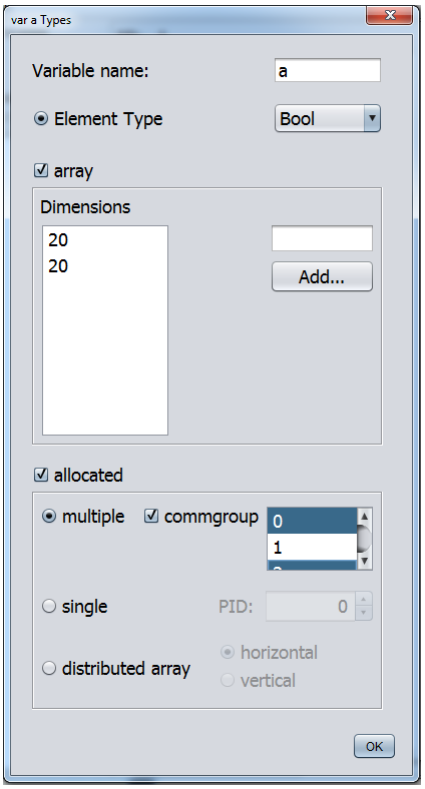


Figure 3.5: Vidim's type panel.

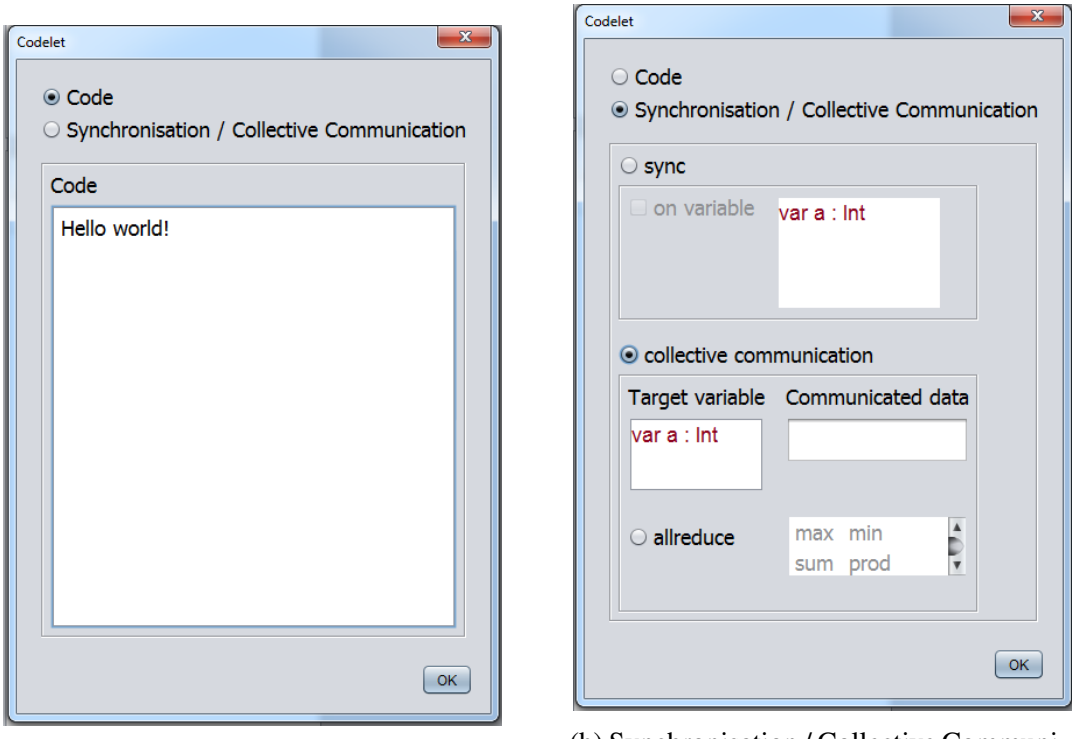
- Which processes the variable is allocated to
  - *multiple* allocation assigns the variable to *all* processes.
  - *multiple* with *commgroup* allows the programmer to assign the variable to any subset of processes.
  - *single* allocation assigns the variable to a single process, specified by the programmer.
  - *distributed array* allocation is only available when the variable is an array. It partitions the array *horizontally* or *vertically*, based on the chosen option, into a number of blocks equal to the number of processes and distributes it evenly across all of them.

When the programmer assigns the variable to a process using the type panel, it automatically appears in the local memory area of this process. In this way, the programmer can always easily identify where any particular piece of data is stored. What is more, using graphical controls is arguably quicker to learn and more intuitive for declaring and specifying the usage of variables when compared to learning textual language syntax.

3.3.3 Process Tasks

In Vidim, tasks executed by the processes are described via textual code, which the programmer enters into the code areas. Vidim allows the programmer to split the code associated with each process into *codelets*, which the programmer describes separately. There are two types of codelet, defined in Vidim:

- **"code"** codelets, which the programmer uses to specify tasks by typing textual code,
- **"synchronisation/collective communication"** codelets, which the programmer uses to specify explicit synchronisations and collective communications by using graphical controls provided by Vidim. These are described separately in Section 3.3.4.



(a) Code

(b) Synchronisation / Collective Communication

Figure 3.6: Vidim's codelet panels for both types of codelets.

To define a new code codelet, the user right-clicks on the code area of the respective process. As a result a new panel, called the *codelet panel*, appears and the programmer then enters the associated code in a text area in that panel. Figure 3.6a shows the codelet panel for the code codelets.

The concept of codelets assists the programmer in two ways:

- It allows them to visually separate distinct tasks.

- It allows them to clearly specify explicit synchronisation points and collective communications.

Figure 3.7 shows implemented Vidim tasks split into codelets.

3.3.4 Synchronisations & Collective Communications

As discussed in Section 3.3.3, explicit synchronisations and collective communications in Vidim are defined as separate codelets by the programmer. In contrast to the code codelets, the synchronisation and collective communication codelets are specified with the use of graphical components provided in the codelet panel. They are shown in Figure 3.6b.

For synchronisations, selecting the `sync` radio button *without* a variable defines a synchronisation on all data updates in the program until this point. Selecting a variable in the list defines a synchronisation on *this variable* - its value can be guaranteed only after the synchronisation has completed. All data which has been declared in the program is automatically displayed in the list, which assists the programmer, as all they need to do is select the desired variable.

For collective communications, the programmer needs to specify the following:

- The target variable. This is the variable which will hold the communicated data after the collective communication has completed. As with the synchronisations, the programmer selects the target variable from a list which is automatically populated with the variables defined in the program.
- The communicated data. This can be any allocated, global or auxiliary variable defined anywhere in the program. The programmer enters its name in the specified text field.
- The type of collective communication. The current implementation of Vidim provides the following collective communications:
  - *allreduce*

By providing special types of codelets for the synchronisations and collective communications, Vidim allows the programmer to define them in a distinct and clear way. This is important, as using these types of operations is not only critical for the correct execution of a parallel program, but is also one of the most challenging aspects of parallel programming. To provide this clarity of representation, Vidim automatically displays the synchronisation and collective communication codelets in **bold** and colours them in the colours of the variables involved in them, as seen in Figure 3.7. What is more, as in the case of data allocation, the graphical components provided by Vidim allow the user to quickly define the codelets by *selecting* their appropriate constructs, as opposed to *coming up* with them.

3.3.5 Code Management

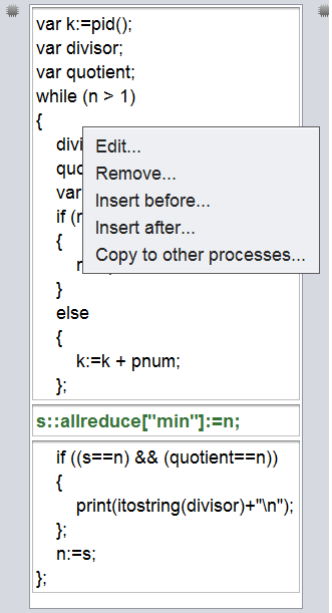


Figure 3.7: Code split into codelets in Vidim, together with the code management pop-up menu.

In addition to adding new codelets, Vidim allows the programmer to manage the code areas in a number of additional useful ways. When the programmer right-clicks on an existing codelet, a pop-up menu with five options, specifying the ways to manage the codelet, appears. It can be seen in Figure 3.7. The options are

- **Edit...**
  This option allows the programmer to edit the codelet. It prompts the codelet panel (Figure 3.6) to appear filled with the current codelet information and allows the programmer to edit it in there.
- **Remove...**
  This option allows the programmer to delete the selected codelet. When pressed, a dialog asking for confirmation of the delete operation appears.
- **Insert before...** and **Insert after...**
  These options allow the programmer to add a new codelet immediately before or after the selected codelet. This is useful in cases when new codelets need to be added between existing ones.
- **Copy to other processes...**
  This option allows the programmer to automatically copy the selected codelet to *all* other processes in the program. This is extremely useful in cases when a large number of processes execute the same tasks, as it allows the programmer to define them in only one place and then copy them over via a simple click.

3.4 Sequential Constructs

In addition to the parallel program constructs provided by Vidim, the language also allows the programmer to define sequential ones, which are available to all processes. These are *global variables* and *functions*. The programmer defines both in an area on *the right-hand side* of the screen, shown in Figure 3.8.

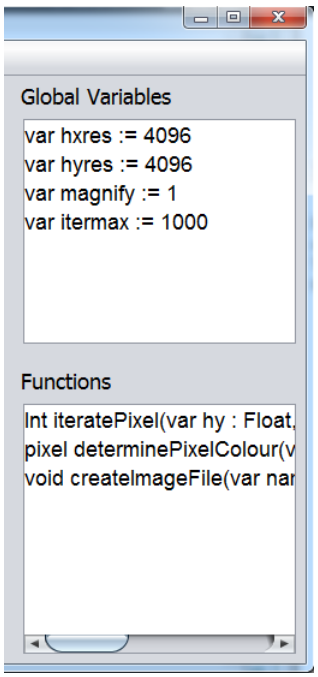


Figure 3.8: Vidim's sequential constructs' area.

3.4.1 Global Variables

Global variables are variables which are available to *all* processes. In other words, they are automatically allocated to all local memories, and processes can access them without performance penalty, as the access requires no communication. The programmer need not specify the element type or allocation for them, but simply a name and value, from which the type is inferred. They are useful for the definition of auxiliary variables. What is more, while their value can be modified by the program, they can be used for the definition of program constants which are used by all processes.

To define them, the programmer needs to right-click in the global variables area, shown in Figure 3.8, similarly to the way they do for allocated variables. They then enter the name and value in a newly displayed panel, called *global variable panel* and shown in Figure 3.9. The new variable automatically appears in the global variables area and can be edited via a right-click, which prompts the global variable panel to appear again.

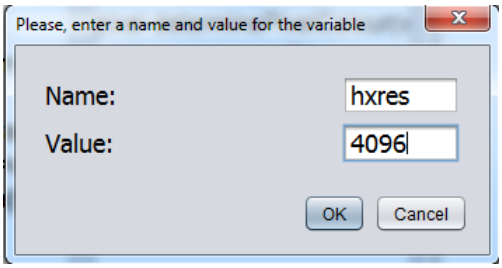


Figure 3.9: Vidim's global variable panel.

3.4.2 Functions

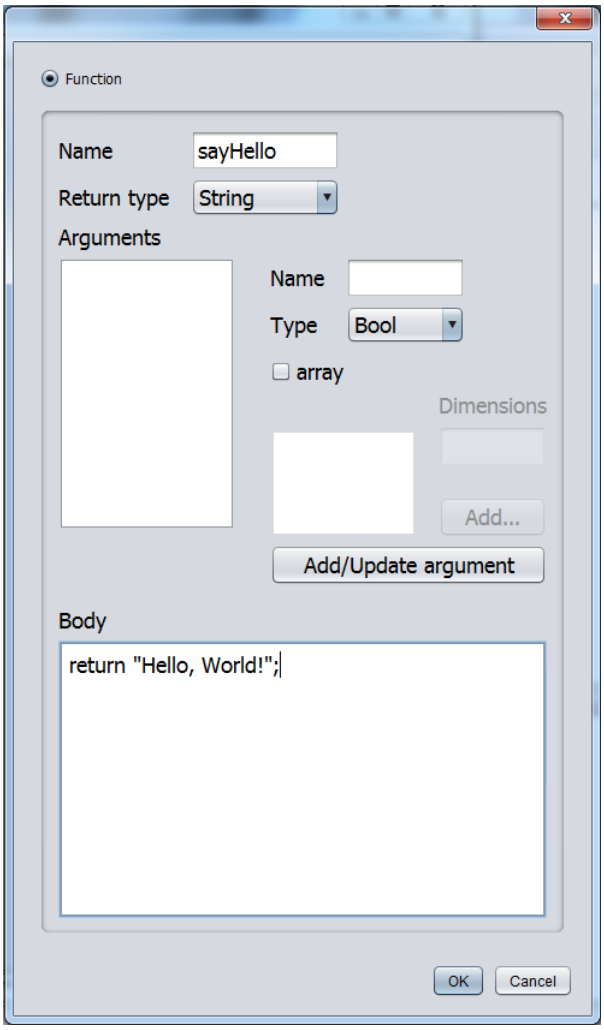


Figure 3.10: Vidim's functions panel.

Vidim allows the programmer to define *sequential* functions, which can be called by all processes. As with the other program constructs, the programmer does this via a right-click in the respective area. In this case this is the functions area, which is shown

in Figure 3.8. A panel, called the *functions panel*, appears and allows the programmer to define the function. The functions panel is shown in Figure 3.10.

Functions are an extremely useful construct in the language, as they allow for functional modularity and reusability of code. As with the other parallel and sequential constructs in Vidim, the graphical components provided by the language assist the programmer in defining them correctly, as they include controls for each of the function's components. They are

- Function's name.
- Return type, which is selected from a drop-down menu as seen in Figure 3.10.
- Arguments. The programmer can add as many arguments as required by specifying their name, type, whether any of them is array and if so, what are its dimensions.
- Code.

3.5 Summary

This chapter defined Vidim - a new visual parallel programming language, based on the PGAS programming model. It uses a graphical representation of high-level parallel program aspects, namely processes and local memories, to add another dimension, which represents parallelism, to traditionally linear textual code. It does this by visualising processes, together with their local memories and code areas, next to one another on the screen and allowing the programmer to specify the tasks to be executed by each process individually by entering textual code in their respective code areas. It also allows the programmer to allocate variables, which appear automatically in the local memories of their respective processes.

In addition, Vidim provides a set of program constructs, both parallel and sequential, which are used by the programmer to define different aspect of their parallel program with the use Vidim's graphical controls. The language automatically customises the visual appearance of some of these aspects once they are defined, to allow the programmer to easily keep track of them. The program constructs are:

Parallel constructs:

- The number of processes
- Allocated data
- Processes tasks, which can be split into two types of *codelets*:
  - "Code" codelet, which the programmer uses to enter tasks in the form of textual code.

– "Synchronisation/collective communication" codelet, which the programmer uses to specify explicit synchronisations and collective communications with the use of Vidim's graphical controls.

Sequential constructs:

- Global variables
- Functions, accessible by all processes

The use of visualisation and graphical controls in Vidim aim for three things:

- to provide a better and more intuitive representation of a parallel program than those provided by purely textual languages,
- to assist the programmer in defining some important aspects of their parallel program by selecting the options provided by the graphical controls, as opposed to using text,
- to prevent some small errors, which occur in textual programming, by detecting and enforcing some dependencies between the constructs of the parallel program.

As will be seen in the following chapter, Chapter 4, the Mesham programming language, presented in Chapter 2, is used as an intermediate representation in the compilation chain of a Vidim parallel program. Therefore, for ease of implementation, the textual code which programmers use to specify tasks in Vidim is also Mesham, as it can be compiled directly with the rest of the program.

Chapter 4

System Implementation

In order to evaluate the feasibility and value of the visual parallel programming approach defined in Chapter 3, a graphical system to enable parallel programming in Vidim was developed.

The current chapter presents the system's implementation, starting from its design and general structure and continuing by describing each module in detail. Along the implementation description, the chapter also includes discussion on the reasons behind the design and development decisions.

4.1 System Structure

Vidim's system enables programmers to perform the full programming process - from designing and implementing their parallel program in Vidim to compiling it into an executable. It achieves this by passing the program's visual representation, as defined in Vidim, through a chain of system components which turn it into an executable through a series of intermediate representations. Figure 4.1 shows the full system structure. As seen in it, the system consists of the following components

- **The GUI.** This is a visual interface, which allows the programmer to design and implement their parallel program in Vidim.
- **The XML Representation.** This a textual representation of the parallel program in the form of an XML file.
- **The Translation Engine.** This is a back-end tool which translates the parallel program into Mesham textual code.
- **The Mesham Representation.** This is an intermediate representation of the parallel program, which consists of valid Mesham textual code.
- **The Mesham Compuler: mcc.** This is the existing Mesham compiler, which translates the Mesham representation into an executable.

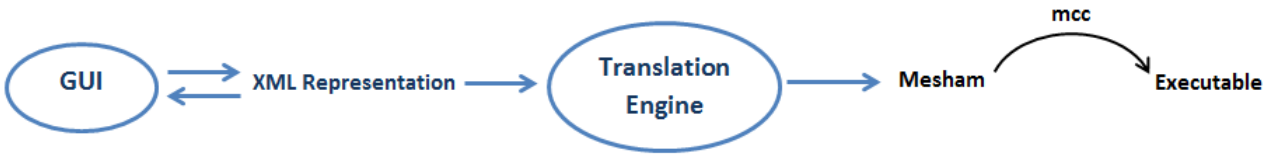


Figure 4.1: Vidim system design.

The components coloured in **blue** in Figure 4.1, namely the GUI, the XML Representation and the Translation Tool, are the ones which were implemented during this project. The components coloured in black, namely the Mesham intermediate representation and compiler, were already existing upon the project's commencement.

Mesham as Intermediate Representation

The higher level of abstraction that Mesham provides, together with its flexibility and maturity, make it a suitable choice as an intermediate representation for Vidim. The visual constructs used by Vidim are easily matched to Mesham's types and the expressiveness of Mesham enables the generality, flexibility and level of abstraction that could make Vidim more intuitive and accessible.

Development Deliverables

Due to its high level of GUI development support and portability across platforms, **Java** was the programming language of choice for this project. In addition, it naturally enabled the use of the object-oriented paradigm, which allowed implementing different constructs of the parallel program as separate objects. This is illustrated in Section 4.2.

The developed system consists of *three* separate, self standing Java projects, namely

- GUI
- Engine
- Library

As their names suggest, the GUI and the Engine projects implement the GUI and the Translation Engine system components respectively. In particular, the GUI project implements Vidim's visual interface and allows the programmer to save their parallel program in XML format. The Engine reads the XML file and translates it into Mesham source code. Thus, the system uses three separate representations of the parallel program:

- Visual representation. This is Vidim's representation which is used by the programmer to architect, describe and implement their parallel program.

- XML Representation. This is the intermediate representation, which sits between the GUI and the Translation Engine. It allows for the decoupling and independence of these two components, which helps their implementation and maintenance. In addition, it allows the programmer to save and re-open their Vidim parallel programs.
- Mesham Representation. This is the intermediate representation, which allows the use of Mesham's existing compiler for the compilation of the parallel program into and executable.

To translate between the three representations, the system uses an *internal abstraction* of the parallel program, which is implemented in the Library project. The Library was developed as a separate project, since the internal abstraction is used by both the GUI and the Engine projects. In this way code replication was avoided and maintainability of the projects was improved. In addition, the Library also implements the modules which write and read the XML files, as well as some global constants which are used by both the GUI and the Engine.

In the rest of the chapter, the implementations of the three projects are described in further detail with focus on their most interesting, challenging and useful aspects.

4.2 Internal Parallel Program Abstraction

The purpose of the back-end components of Vidim's system is to translate the visually represented parallel program into an executable. As discussed in Section 4.1, these separate components produce two intermediate textual representations. In order to enable the system's components to communicate with one another, an *internal abstraction* of a Vidim parallel program was defined and implemented in the Library project.

The internal abstraction roughly views a Vidim parallel program as *a collection of its parallel and sequential constructs*, as described in Sections 3.3 and 3.4 respectively. Table 4.1 presents the constructs which comprise the program's internal abstraction. Each of them is implemented as a separate Java object. For each construct, table 4.1 also displays the attributes which describe it.

In short, a Vidim's parallel program is internally represented in the system as a collection of:

- An integer to specify the number of processes
- An array of Processes
- An array of Allocated Variables
- An array of Global Variables
- An array of Functions

Program construct	Attributes
The number of processes	
The **Processes**	• ID. • An array of **Variables** allocated to it. • An array of **Codelets**, describing the code it executes.
The **Allocated Variables**	• Name. • Element type, as described in Section 3.3.2. • Allocation, described as `single`, `multiple` or `distributed array`, outlined in Section 3.3.2. • An array of the IDs of the processes to which it is allocated. • Whether the variable is an array and if so, its dimensions. • Colour.
The **Global Variables**	• Name. • Value.
The **Functions**	• Name. • Return Type. • An array of variables, which are the function's arguments. • The code executed by the function.

Table 4.1: Internal abstraction of a Vidim parallel program.

XML Representation

The Library project also implements the XML reader and writer. The writer is used by the GUI in order to save the parallel program implemented in it as an XML file, while the reader is used by both the GUI to open a saved program, and the Translation Engine to compile the Vidim program into Mesham textual code.

The structure for the XML intermediate representation can be seen in Appendix A. It allows for the independent development and maintenance of the GUI and the Translation Engine. This modularity is not only useful from software development point of

view, but also provides some conceptual flexibility for potential future developments in the system. For example, while Mesham was chosen as an intermediate representation in this project, future developments might choose a different approach to compiling the Vidim programs into binaries. Having the XML representation between the GUI and the Translation Engine means that such developments could change or replace the Translation Engine without affecting the GUI. In addition, it allows for the easy saving of the Vidim programs in compact text files.

XML was chosen as the format of this representation because of its wide popularity and clear structure, the compactness of the XML files and the wide existing support for XML parsing. The current system used Java's Simple API for XML (SAX) parser [Oracle, 2013].

4.3 GUI

The GUI was implemented in the GUI project. While the Library project implements the very important internal abstraction of a Vidim parallel application, it is the GUI that enables the development of one in the first place. Figure 4.2 shows the full visual interface implemented in the GUI. In addition, the GUI also implements all graphical control panels, which are shown for illustration in Chapter 3.

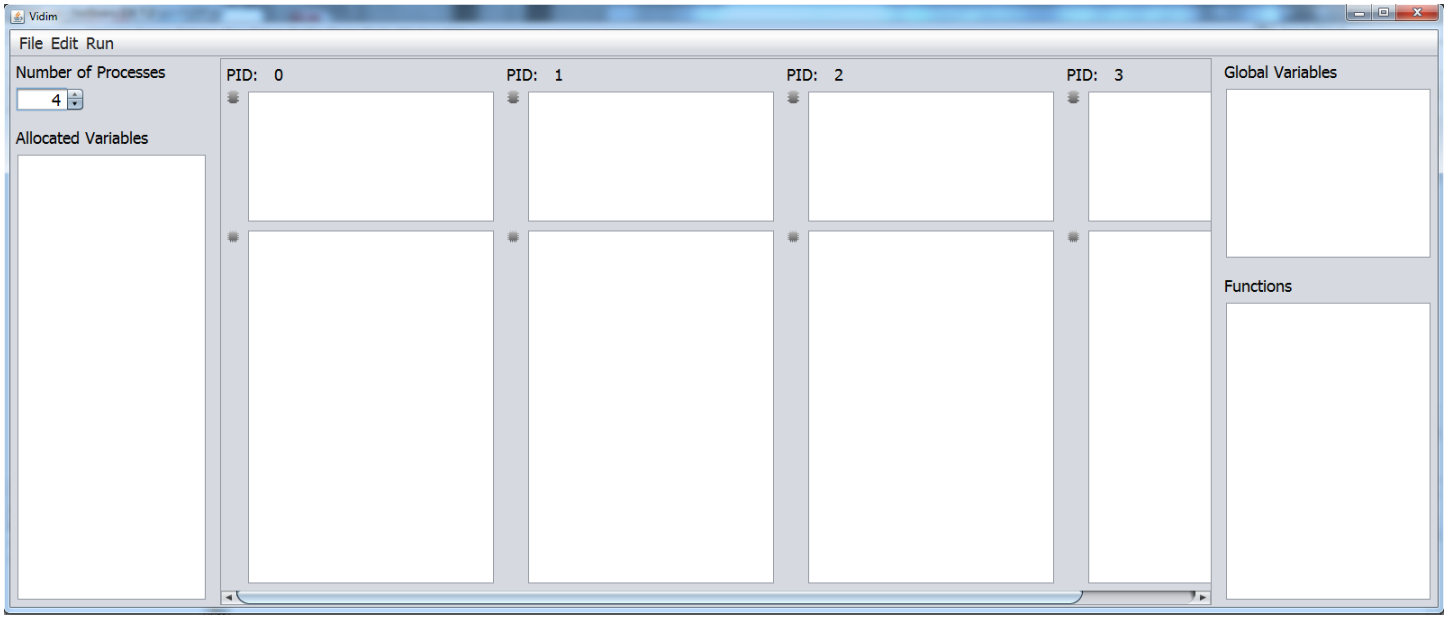


Figure 4.2: Vidim's visual interface.

Like the rest of the system, the GUI was developed in Java with the use of the Swing toolkit. Its design follows the structure outlined in the language definition, by splitting the screen three areas:

- Left-hand side area, in which the programmer specifies the parallel aspects of their application.
- Middle area, in which processes, together with their local memories and code areas, are displayed. The programmer can then add tasks directly in the code areas.
- Right-hand side area, in which the programmer specifies the sequential aspects of their application.

These areas are developed in separate Java classes, which represents their conceptual separation and makes each of them easily manageable.

Each of the graphic control panels shown in Chapter 3 is also implemented in its own class. They are used both for the specification of a new parallel program construct and for the editing of existing ones. Thus, the GUI implements mechanisms to allow them to automatically get filled with the properties of the respective constructs, when they are used for editing. For example, if the type panel is used to update an already defined variable, it will appear with the variable name and type components already filled.

Furthermore, the GUI's implementation takes care to enable enough connectivity between components in order to detect dependencies between the program constructs. This is important, as it allows the implementation of Vidim's graphical controls, discussed in Chapter 3, which guide the programmer in simple decisions and prevent errors. For example, the type panel, which allows the addition and editing of allocated data, always knows the number of processes, so it does not allow the programmer to allocate data on non-existing processes. It also does not allow the distribution of variables, which are not arrays. Another example are the codelet panels, which allow the programmer to define and edit codelets. They always know what allocated variables have been defined in the program and allow the programmer to select from them in synchronisation/collective communication tasks.

Finally, the GUI implements toolbar menus, which allow the programmer to save, open and compile programs into Mesham textual code.

1. When the programmer selects the Save menu, a file chooser appears to allow the programmer to select a directory and a name for the saved program. Then, the internal abstraction is passed to the XML writer, which produces an XML representation of the parallel program and saves it in a file.
2. The GUI allows programmers to open parallel programs, which were previously saved as XML files, for editing. After selecting the Open menu, the programmer can choose a parallel program in a file chooser. The GUI then uses the XML reader to get the internal abstraction of the parallel program and display it.
3. When the programmer selects the Compile menu, the GUI automatically saves the parallel program in an XML file, using the same implemented function as point 1. It then calls the Translation Engine, and passes it the newly saved XML representation. The Translation Engine then produces the Mesham textual code.

These capabilities illustrate the usefulness of the XML representation.

4.4 Translation Engine

The Translation Engine was implemented in the Engine project. It consists of a single function, which uses the XML reader from the Library project to retrieve the Vidim parallel program in its internal abstraction form. It then translates it into Mesham textual code, which it writes in a separate source file.

The translation is easily enabled by the close mapping of Vidim's program constructs, as described in Chapter 3 and Mesham's types and keywords, as briefly discussed in Section 2.1.3. Table 4.2 shows this mapping.

Constructs in Vidim	**Mapped Types and Keywords in Mesham**
Allocated Data:	
• Element types	`Bool, Int, Double, Char, String, Float, Short, Long, File`
• *pixel* element type	`record["r",Int,"g",Int,"b",Int]`
• Array and array dimensions	`array[`*type*`,`$d_1, d_2, ..., d_n$`]`
• Process allocation	`multiple[], multiple[commgroup[]], single[], horizontal[], vertical[]`
Code executed by Process *id*	`proc` *id*`{`*code*`};`
Explicit synchronisation:	`sync` *var*`;`
Collective communications:	
• allreduce	`allreduce[`*operation*`]`
Function	`function` *return_type* *name*`(`*arguments*`)`

Table 4.2: Mapping of Vidim's program constructs to Mesham's types and keywords.

In other words, the system's use of Mesham as an intermediate representation not only allows the use of Mesham's compiler for the production of binaries, but also enabled the implementation of the Translation Engine to take advantage of the way Mesham uses its type libraries to abstract away parallel language complexities.

4.5 Summary

The current chapter presented the implementation of Vidim's visual system. It is an end-to-end system which allows programmers to develop Vidim parallel applications and compile them into executable files. It consists of three main components

- GUI,
- Translation Engine,
- Mesham compiler,

of which the GUI and the Translation Engine were implemented during this project, while the Mesham compiler was already existing. Two parallel program intermediate representations are used, namely XML representation between the GUI and the Translation Engine and Mesham textual code between the Translation Engine and the Mesham compiler. In addition, the two developed system components use an internal abstraction of the parallel program to communicate with one another. Both the XML representation and the internal abstraction were defined in the course of the system implementation.

The system was developed into three separate Java projects:

- GUI, which implements the GUI
- Engine, which implements the Translation Engine
- Library, which implements the internal abstraction, as well as the XML reader and writer.

The overall system design, as well as particular design decisions concerning each component, were discussed in this chapter. Considerations behind them included both the system's desired functionalities and its maintainability.

Chapter 5

Results & Evaluation

The aim of this project is to demonstrate that the use of visual programming languages can result in parallel program representations which are more intuitive than textual code and make parallel programming more accessible. To clarify and specify this goal, a set of evaluation criteria was identified in Section 2.3. It is the first tool used in the evaluation of the produced visual parallel programming language. The second tool is a set of two case studies, which were implemented in Vidim and compared to their respective implementations with some of the existing and widely used textual parallel programming frameworks, namely MPI, OpenMP and Mesham.

The current chapter presents and evaluates the results achieved in this dissertation. It starts with a detailed presentation of the case studies and their particular parallel challenges. It then describes their implementations in Vidim and discusses the way these challenges are addressed by the visual language. Finally, it revisits the evaluation criteria defined in Section 2.3 and uses the results presented by the case studies in order to evaluate Vidim against them.

5.1 Case Studies

The two case studies used for the evaluation of the new visual parallel programming language are parallel computation of **Prime Factorisation** and drawing of the **Mandelbrot Set**. They were chosen for the interesting parallel aspects of their computation, for their popularity, for the conciseness of their code and for the availability of their implementations with other parallel programming frameworks.

In should be pointed out that in addition to the two case studies presented in this section, two more were considered for evaluation. Those were parallel Matrix Multiplication and Image Reconstruction. After careful analysis of the challenges behind them, it was found that they were very similar to those of the two implemented case studies. For example, data distribution as a challenge in the Image Reconstruction case study is also a challenge in the Mandelbrot Set case study. As such, it allows for two separate

implementations of the Mandelbrot Set case study and a comparison between them. Consequently, not only are Vidim's mechanisms for solving the challenges of the two case studies the same, but focusing on the Mandelbrot Set case study allows for an interesting comparison of the different ways in which it can be implemented in Vidim. Therefore, it was decided that the two case studies, Matrix Multiplication and Image Reconstruction, be omitted from the final dissertation to avoid redundancy and maintain conciseness.

The current section describes the two implemented case studies, Prime Factorisation and Mandelbrot Set, in detail by presenting their algorithms and identifying their particular parallel aspects.

5.1.1 Prime Factorisation

The prime factorisation parallel program outputs the prime factors of a given positive integer by iterating through a set of prime numbers, starting from 2 in increasing order, and continuously dividing the input by the primes which are its divisors, until it becomes one.

Algorithm

The algorithm used in this case study is the following:

Algorithm 5.1 Prime Factorisation

Require: input n

1: **while** $n > 1$ **do**
2: $k = 1$
3: $divisor = k^{th}$ prime number
4: **if** n is divisible by $divisor$ **then**
5: divide n by $divisor$
6: print $divisor$
7: **else**
8: increment k
9: **end if**
10: **end while**

It is an extremely straight-forward algorithm, which can be easily parallelised by allowing separate processes to check if different prime numbers are factors of the input in the same time. For example, if n processes execute the algorithm in parallel, in the first iteration of the while loop, they will test the first n prime numbers concurrently.

Parallelising the algorithm requires the following changes:

- On line 2., initialise k to the process id

- On line 8., increment k by the number of processes

Depending on the programming model chosen for the implementation of the parallel algorithm, some interesting considerations emerge.

Interesting Parallel Aspects

- All processes start executing the algorithm concurrently. It has no sequential portions.
- On distributed memory architectures, each processor needs to know the value of the input number, thus it needs to be distributed and stored in each local memory.
- Processes need to coordinate at the end of each iteration of the while loop, in order to determine the value of the input, after it has been divided by the prime factors identified by each of them.
- On distributed memory architectures, this would involve a collective communication, in which all processes communicate and reduce their respective values. It is an *allreduce* communication.
- On shared memory architectures, processes write the values of the divided input directly into memory. What is more, the order in which they do this does not affect the correctness of the program, as all processes divide by different prime factors and the number needs to be divided by all of them in any order. Therefore, in this particular case, no explicit synchronisations are needed in a shared memory implementation.

5.1.2 Mandelbrot Set

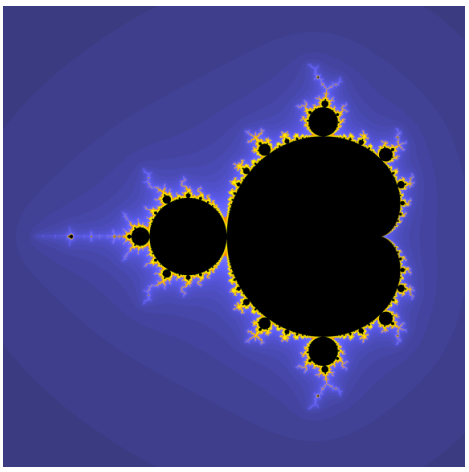


Figure 5.1: An image of the Mandelbrot set as drawn by the implementations presented in Section 5.2.2, when executed on Morar. Size: 4096 x 4096 pixels.

As its name suggests, the Mandelbrot set parallel program draws an image of the Mandelbrot set. The dimensions of the image are set as a parameter in the program and can be changed. Figure 5.1 shows the image produced by the implementations described in this chapter.

Algorithm

The Mandelbrot set case study uses a popular iterative algorithm to determine whether a given point belongs to the set or not. Based on the number of iterations performed for each point, it determines the colour of the respective pixel in the final image.

The high level algorithm is the following:

Algorithm 5.2 Mandelbrot

Require: input image dimensions $hxres$ and $hyres$

```
1: declare image[hxres][hyres]
2: for hx from 0 to hxres do
3:    for hy from 0 to hyres do
4:       iteration = iteratePixel(hx, hy)
5:       image[hx][hy] = determinePixelColour(iteration)
6:    end for
7: end for
8: createImageFile(image)
```

where

- `iteratePixel`(hx, hy) takes the coordinates of the image point and performs an iterative algorithm, which determines if the point is in the Mandelbrot set. It returns an integer, which is used to determine the colour of the given point in the output.
- `determinePixelColour`($iteration$) uses the integer returned by `iteratePixel` to determine the colour of the respective pixel.
- `createImageFile`($image$) draws the image of the Mandelbrot set in a `.ppm` file.

Since these operations are implemented as separate self-standing *sequential* functions, they do not impact the parallelism in the case study and thus, for conciseness, further details about them is omitted from the current chapter. Nevertheless, their implementations in **C** can be found in Appendix B.

Loops are a natural source of parallelism in many applications and this is the case here too. In this case, parallelising the loop consists of splitting the image into rows (or columns) and letting the processes perform the computations in parallel - each on its respective portion of the image. What is more, parallelisation is particularly appropriate

in this use case, as all of the loop's iterations are *entirely independent* - all information needed to determine whether a point belongs to the Mandelbrot set or not is its coordinates.

Interesting parallel aspects

- Since drawing the image into a file is a sequential operation, irrelevant of the parallel programming model used, one process should have access to the whole image in the end.
- On shared memory architectures, all processes have access to the whole image at all times. Despite all of them accessing the image concurrently during the parallel loop, each process accesses a different area of it, which means that no explicit synchronisation is needed.
- On distributed memory architectures which are programmed with the message-passing programming model, all portions of the image need to be explicitly communicated to a single process after the parallel loop has finished executing in order for this process to be able to draw it into a file.
- On distributed memory architectures which are programmed with the PGAS programming model, it is also possible for processes to write directly into a single process's memory during the parallel loop. This would require explicit synchronisation after each update of the image, in order to ensure that the remote writes have completed.

5.2 Implementation

After Section 5.1 introduced the case studies used for the evaluation of Vidim, the current section presents their implementations in Vidim, discusses how it handles their parallel challenges and compares them to implementations with textual parallel frameworks.

5.2.1 Prime Factorisation

Figure 5.2 shows the Prime Factorisation case study as implemented on 4 processes in Vidim. Considering the parallel aspects of the program, the implementation illustrates the following useful features of the visual language.

Firstly, in separate code areas it clearly displays what tasks are executed by each process at all times. In this way, it adds a dimension to the program representation, through which processes' tasks can be laid out by the programmer and represented literally *in parallel*. This is useful, as it allows the programmer not only to think about and specify

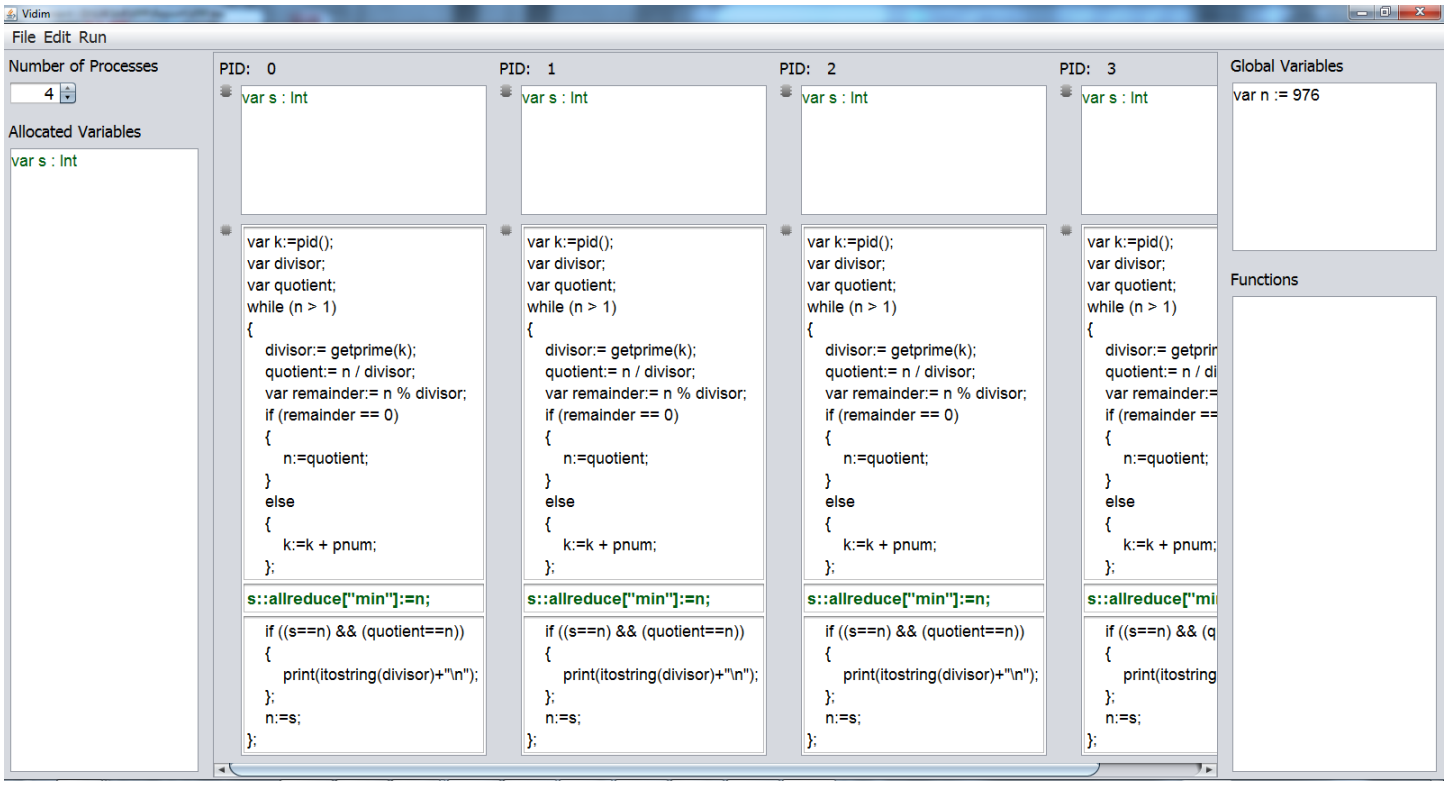


Figure 5.2: **Prime Factorisation** implemented on 4 processes in Vidim.

the tasks executed by each process separately, but also to clearly see what all of them will be executing concurrently.

In this particular case study, all processes execute exactly the same tasks. As discussed in Section 3.3.3, Vidim conveniently allows the programmer to enter them in the code area of one process only and then automatically copy them over to the code areas of the other processes.

Secondly, Vidim allows the programmer to specify the collective communication, in this case `allreduce`, separately and displays it clearly in a separate codelet, in bald font, coloured in the colour of the variable participating in it. This is important, as the correct placement of collective communications is crucial for the execution and correctness of a parallel program and distinguishing them in the program's representation helps the programmer achieve this. What is more, they are blocking and the alignment of their calls in all code areas, together with their distinct appearance, enhances their representation as a "barrier" - the programmer clearly sees the point which all processes need to reach before continuing with their individual executions.

Finally, Vidim allows the programmer to clearly see where they have allocated the program's data. The variable `s`, which is used to collect the result of the `allreduce` communication on all processes, is clearly declared as an integer and allocated to the local memories of all processes - it is displayed in the local memory areas of all of them. Furthermore, the ability to specify the input number which is factorised, `n`, as a global

variable allows the programmer to automatically allocate it to all processes and assign it an initial value.

5.2.2 Mandelbrot Set

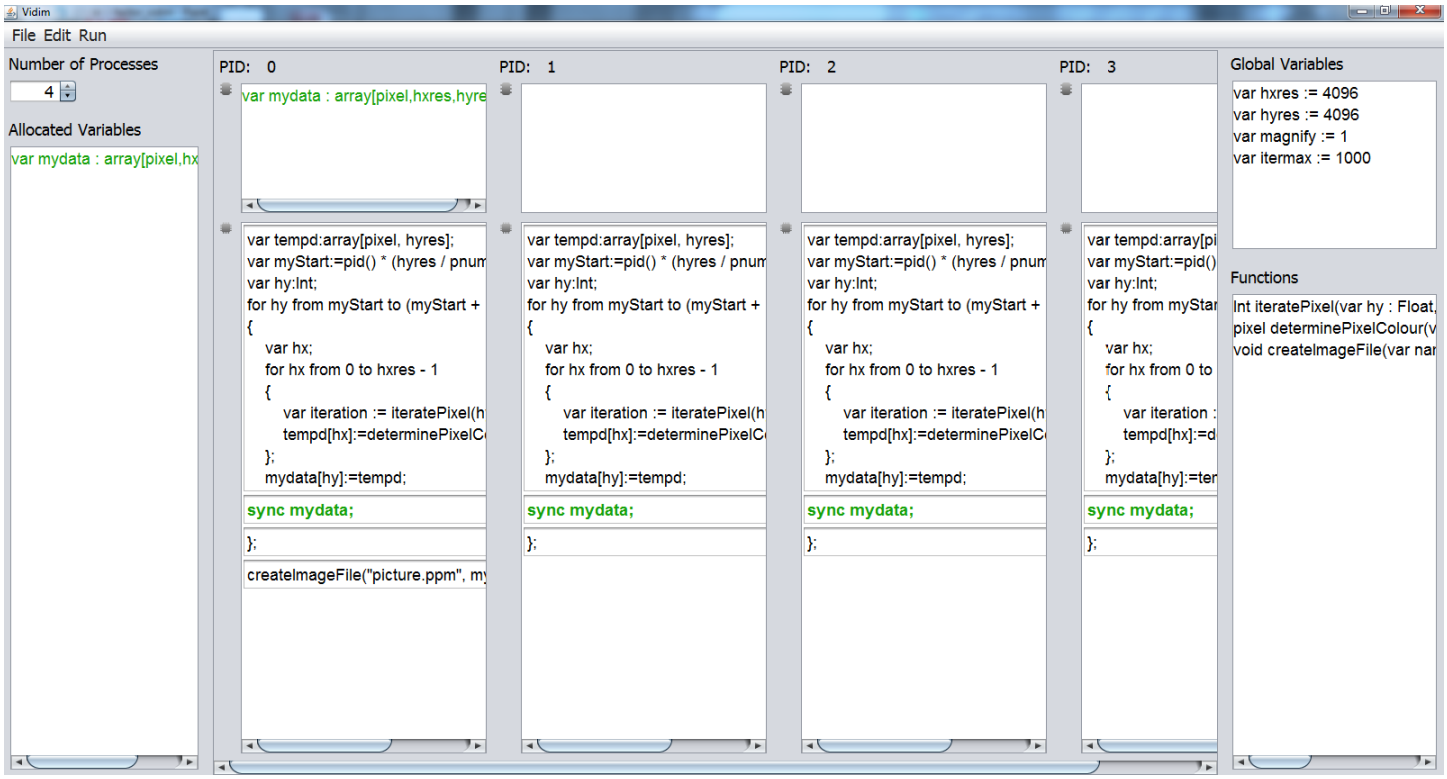


Figure 5.3: First implementation of the **Mandelbrot Set** generation in Vidim on 4 processes.

The Mandelbrot Set case study is particularly interesting, as it can be implemented in two separate ways in Vidim with each implementation illustrating different useful aspects of the language.

- In the first implementation, the double array which represents the image is stored on a single process, say process 0. The rest of the processes use the global address space provided by Vidim's programming model to access their respective portions of the image directly during the parallel computations.
- In the second implementation, the double array which represents the image is split horizontally and distributed among the local memories of all processes. In this way they perform all their operations locally, without any remote accesses. In the end, process 0 uses the global address space to collect all portions of the image and write it into a file.

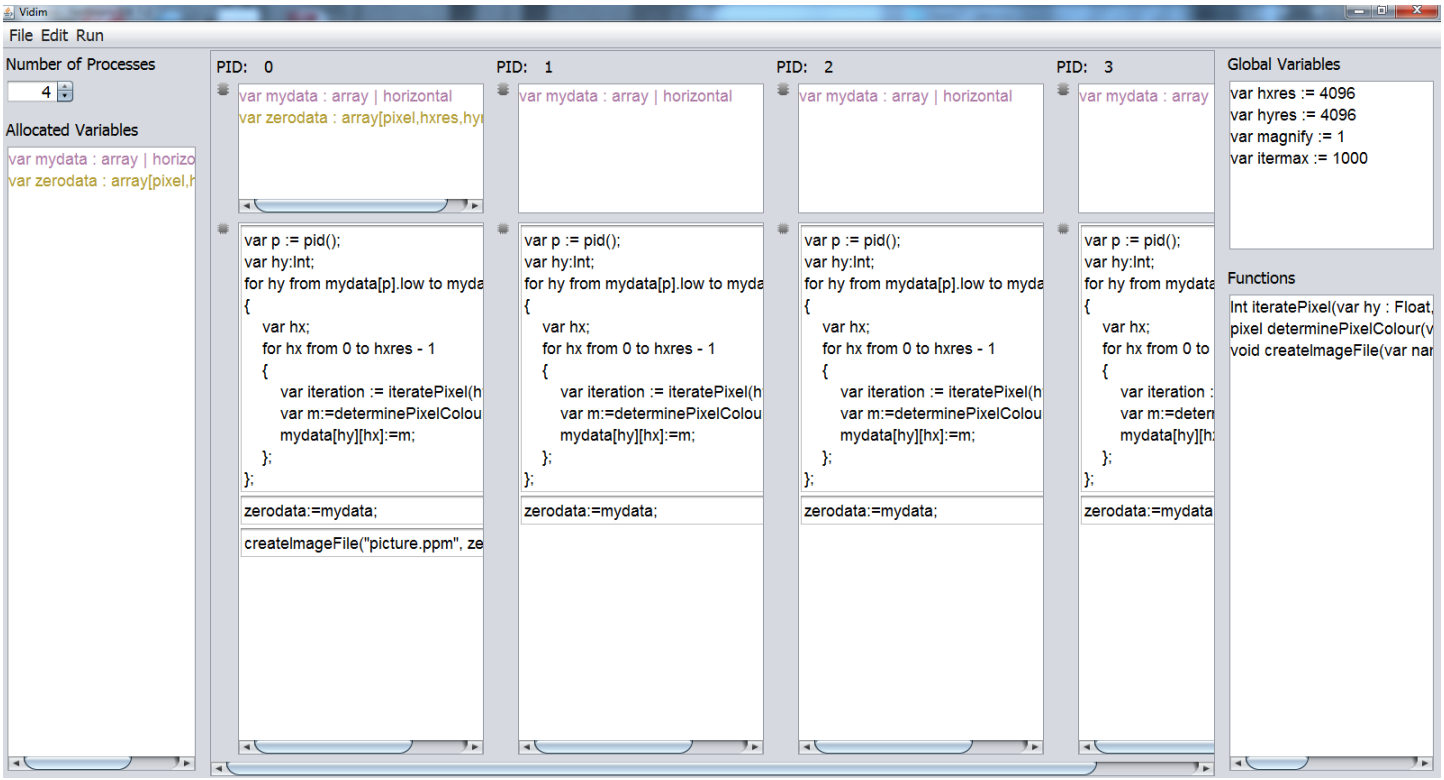


Figure 5.4: Second implementation of the **Mandelbrot Set** generation in Vidim on 4 processes.

Both implementations are presented and discussed in detail in this section. They are shown in Figures 5.3 and 5.4 respectively. While the dynamic system allows the programmer to scroll in the code areas in order to read the code, this is not possible in the screenshots presented in the two figures. Therefore Figures 5.5 and 5.6 are provided to show the full textual codes of the two implementations. Notice that in both implementations, processes 1 to 3 execute the same tasks, thus it is enough to show process 1 only.

The visual representation provided by Vidim allows the observer to notice the main differences between the two implementations at a glance.

Firstly, the two visualisations clearly show the data allocation of the image array, called `mydata`. In the first implementation, Figure 5.3, it is allocated to process 0 only and it appears only in its local memory area. In the second implementation, Figure 5.3, it is split horizontally and distributed across all processes and hence it appears in the local memory areas of all of them with the word "horizontal" next to it.

Secondly, much like with the collective communications shown in the implementation of the Prime Factorisation case study in Figure 5.2, Vidim very clearly displays the explicit synchronisations which are necessary for the correctness of the first implementation of the Mandelbrot Set (Figure 5.3). The synchronisations are needed, as during the loop iterations all processes, but process 0, update `mydata` *remotely*. The synchro-

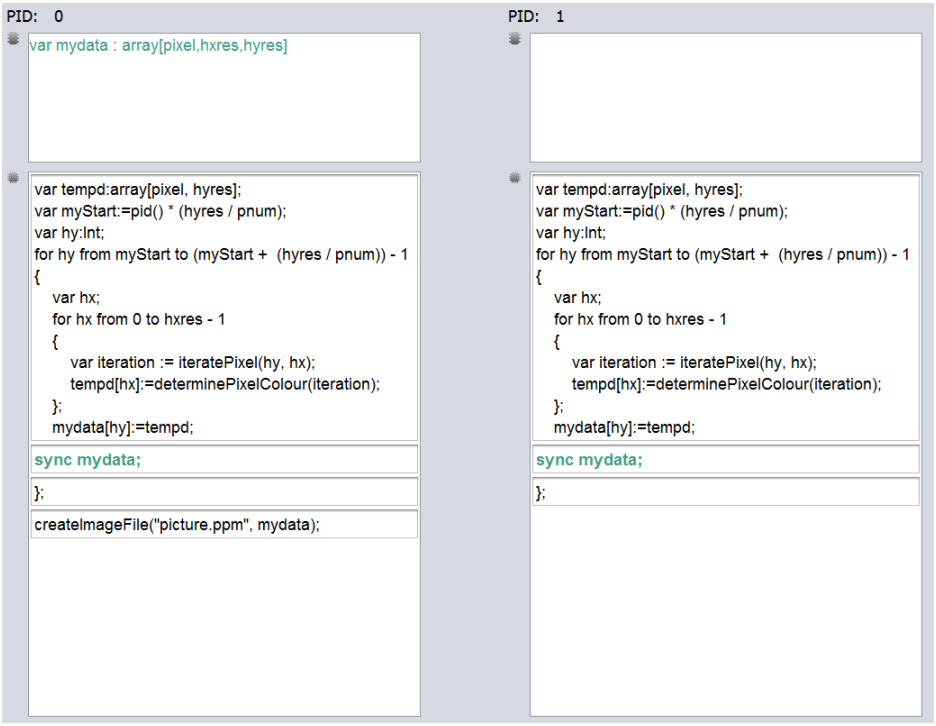


Figure 5.5: Code areas displaying the entire textual code in the *first* implementation of the **Mandelbrot Set** generation in Vidim.

nisations guarantee that these remote accesses have completed. As demonstrated in Figure 5.4, the explicit synchronisations are not needed in the second implementation, since each process updates their local partition of `mydata`. Nevertheless, in order to draw the image into a file, process 0 needs all portions in its own memory in the end. Thus, as clearly shown, `zerodata` is declared, allocated to process 0 and assigned the value of `mydata` after the iterations have completed. This assignment has the effect of a collective gathering communication, but conveniently does not require the explicit coding of one, as process 0 can access all portions of the image directly through the global address space.

Furthermore, there is another, more subtle difference between the data distribution mechanisms employed in the two implementations. In the first implementation, Figure 5.3, all processes access the image, `mydata` on process 0 directly, but need to compute the end coordinates of the portion they should update on their own. This requires a computation of the form

```
var myStart = pid()*(hyres / pnum);
var myEnd = myStart + (hyres / pnum) - 1;
```

on all processes. In contrast, in the second implementation, the automatic horizontal partitioning of the image allows not only the programmer to view the way the processes are partitioned clearly, but also the processes to access their own partitions via an index, for example `mydata[0]`.

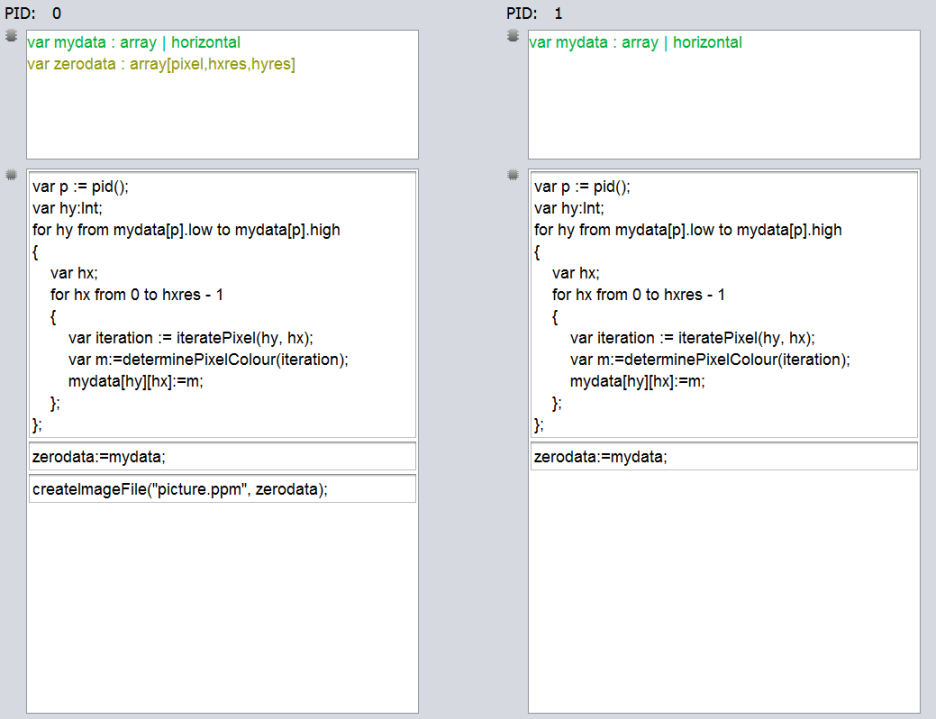


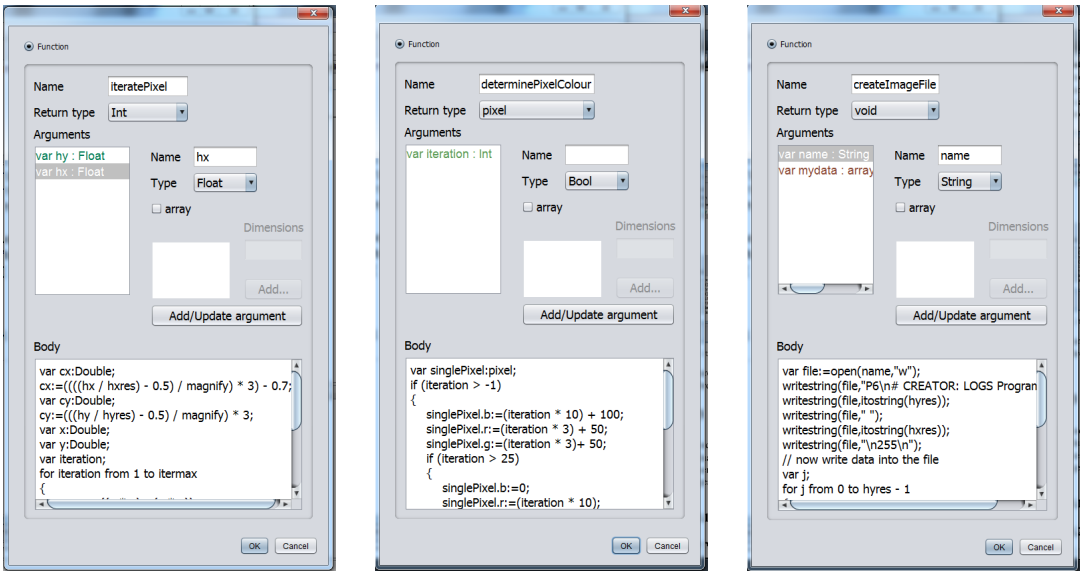
Figure 5.6: Code areas displaying the entire textual code in the *second* implementation of the **Mandelbrot Set** generation in Vidim.

Based on the comparison made between the two implementations, it would appear that due to the features provided by Vidim, the second implementation is easier from programmability point of view. It uses automatic data partitioning and requires no explicit synchronisations between processes. Nevertheless, should other considerations arise, for example performance, which require the Mandelbrot Set case study to be implemented as in the fist implementation, Vidim's visualisations simplify the programming tasks, by providing clear representations of the program's important parallel features.

Additionally, the two Mandelbrot Set implementations illustrate the way sequential functions, available to all processes, are implemented in Vidim. As discussed in Section 3.4.2, functions are displayed in their area on the right-hand side of the screen and can be defined and edited in a separate panel which appears upon a right-click in the area. Figure 5.7 shows the implementations of the three functions used by the Mandelbrot Set case study. The visual controls allow the user to quickly and easily specify and edit a function's name, return type, parameters and operations performed by it.

5.3 Evaluation

In order to achieve the dissertation's goals, it is Vidim's aim to satisfy a set of evaluation criteria, defined in Section 2.3. The current section draws on the results presented in Section 5.2 in order to evaluate Vidim's success in addressing the challenges behind



`iteratePixel()` `determinePixelColour()` `crateImageFile()`

Figure 5.7: Vidim's implementations of the three sequential functions used int he Mandelbrot Set case study.

each criterion. In doing so, it compares the case studies' implementations in Vidim to those in OpenMP and MPI. To enable this comparison, the case studies were implemented in C with OpenMP and MPI during the course of this dissertation. While this section uses only excerpts from these implementations for brevity and conciseness, their full source code can be found in Appendix C.

A brief comparison can also be made with Mesham's implementations of the two case studies which can be found in the "Example Codes" page on Mesham's website in [Brown, 2013]. As discussed in Section 2.1.3 and shown in Table 2.1, Mesham's type oriented approach provides a lot of assistance in parallel programming similar to that in Vidim. The areas in which Mesham is less successful, namely clear synchronisation and data distribution, Vidim improves by distinguishing the visual representation of the explicit synchronisations and visualising the data allocation and distribution in separate memory areas.

5.3.1 Using a General Programming Model

As mentioned many times throughout this dissertation, Vidim uses the partitioned global address space (PGAS) parallel programming model, which combines the high degree of flexibility of MPI with the high levels of programmability of OpenMP. Many of the useful features provided by Vidim are enabled by the use of this model, which is discussed in detail in the following subsections. Examples include direct accesses to remote memories, as seen in the first implementation of the Mandelbrot Set case study shown in Figure 5.3 and automatic data partitioning, shown in the second implementa-

tion of the same case study, Figure 5.4.

5.3.2 Providing Abstraction of Low-Level Communication Details

As discussed in Section 2.1.2, one of the biggest challenges in parallel programming is explicit communication between processes. It is a particular challenge in distributed memory machines, which are traditionally programmed with the message passing programming model, because it requires the coding of explicit message sends and receives on all processes that participate in the communication. Vidim addresses this challenge in a number of ways.

Firstly, it uses the PGAS programming model, in which processes communicate via direct reads and writes into one another's memory and there is no need for explicit message passing. This is demonstrated by the second implementation of the Mandelbrot Set case study, shown in Figure 5.4. After all processes have completed updating their portions of `mydata`, process 0 needs to collect all portions into its own `zerodata` array. In MPI this requires an explicit *gather* communication of the form:

```
MPI_Gather(mydata, own_x*own_y, mpi_pixel_type, zerodata,
   own_x*own_y, mpi_pixel_type, 0, commWorld);
```

in which the programmer needs to specify not only the data which is communicated, but also a number of additional things. The first thing is computing the data's size. Secondly, the programmer needs to consider the data's type and if not elementary, but a structure, they need to define a new type. This requires a lot of preliminary computations and MPI function calls, which the programmer needs to be familiar with. Finally, the programmer needs to consider which processes participate in the communication and define a communicator for it. In contrast, in Vidim, all process can assign their portions of `mydata` directly to process 0's `zerodata`. This is helpful, because it requires

- no explicit computation of the size of the communicated data,
- no need to create a new type for communicating structures,
- no need to consider communicators, as the programmer implements the communication directly on the participating processes.

Vidim's direct remote accesses are also demonstrated in the first implementation of the Mandelbrot Set, shown in Figure 5.3, in which each process updates `mydata` directly during the iterative steps, even though it is stored on process 0.

Secondly, while Vidim's direct remote accesses eliminate the need of explicit message passing, explicit collective communications are still useful in cases when multiple processes need to participate in the communication. Therefore, Vidim also provides explicit collective communications. However, unlike MPI, Vidim abstracts away low-level parameters, such as the data size, type and communicators. Instead, it allows the

programmer to use separate graphical components to easily define the communication, by specifying only the participating variables and operations, as shown in Figure 3.6b. The communication is then automatically displayed in distinguished colours and with bold fonts in the code areas, as shown in Vidim's implementation of the Prime Factorisation case study in Figure 5.2. This is useful, as it helps the programmer to identify and keep track of these communications, which is important as they are blocking and potentially expensive. For comparison, much like the *gather* communication shown above, the *allreduce* communication in the MPI's implementation of the the Prime Factorisation case study

```
MPI_Allreduce(&n, &s, 1, MPI_INT, MPI_MIN, commWorld);
```

requires from the programmer to not only be familiar with the relatively complex syntax of the `MPI_Allreduce` call, but also to explicitly specify a number of low-level parameters.

Finally, unlike textual languages, Vidim provides visual representation of the program's data allocation, which assists the programmer in identifying not only the type of communication they need - be it local, remote or collective, but also which processes need to participate in it.

5.3.3 Providing Clear Ways for Specifying Explicit Synchronisations

Even though the use of the PGAS programming model allows for direct reads and writes into the processes' memories, it raises the issue of synchronising simultaneous accesses by multiple processes to the same data. If two processes try to change the same variable at the same time, which value should be recorded? This brings the necessity of providing clear ways to specify explicit synchronisation points. As discussed in Section 2.1.1, this is a challenge that is commonly found in programming models which use shared address spaces.

Vidim addresses this challenge by visualising the code for each process in separate areas, which allows the programmer to easily identify when multiple processes access the same data, as well as where to position the synchronisations. In comparison, OpenMP requires the programmer to use compiler directives for many of the specifications concerning parallel execution, such as the definition of parallel regions and loops, the specification of local and shared data in the parallel regions and the definition of critical regions which can be executed by only one process at a time. An example of such a compiler directive can be found in the OpenMP implementation of the Mandelbrot Set case study.

```
#pragma omp parallel for default(none) private(hx,hy)
   shared(mydata) schedule(static)
```

This compiler directive is all that is needed to split the shared image, `mydata`, into equal partitions and allow processes to work on it concurrently. This is undeniably simple and is an example of what classifies OpenMP as a relatively accessible parallel framework. Nevertheless, not only can the directive be easily misplaced and hard to find in textual code, but it also requires the programmer to be familiar with its relatively cryptic syntax. For example, the programmer needs to know that `schedule(static)` means "split work equally across processes", as well as what will happen if they do not specify it. In contrast, Vidim's first implementation, shown in Figure 5.3, demonstrates how the programmer can simply program each process to calculate its respective portion of `mydata` directly and can quickly specify the synchronisation points upon its updates.

In addition, as with collective communications, Vidim automatically provides distinguished appearance of the synchronisation code by using bald font and the variable's colours, as seen in Figure 5.3. This allows the programmer to recognise the explicit synchronisations at a glance and is useful for debugging purposes. It also provides a nice visual representation of the blocking nature of the synchronisations, as it makes them stand out as "barriers" up to which all processes need to finish execution before continuing. In comparison, the synchronisation points in an MPI program, while clearly defined through calls to blocking communications or explicit barriers, lack this visual representation and are harder to keep track of amidst the rest of the textual code.

5.3.4 Providing Clear Ways to Distribute Data

Another main challenge in parallel programming is data distribution on distributed memory architectures. This is so not only because it often requires explicit message passing to partition the data, but also because textual programs do not have a way of representing separate partitions.

As demonstrated in the second implementation of the Mandelbrot Set case study shown in Figure 5.4, Vidim addresses this challenge in two ways. Firstly, it provides automatic partitioning and data distribution of arrays. The programmer can easily specify it using the graphical components provided in the variable's type panel, shown in 3.5. Each process can then directly access its partition locally, which is easy to program.

Secondly, it clearly displays the partitions, as well as other allocated data in the local memory areas of the processes. This allows the programmer to to know at all times how data was allocated and distributed.

In contrast, in MPI's implementation of the Mandelbrot Set case study, the array representing the image is partitioned across processes into local blocks, called `mydata`, each of dimensions `[own_x, own_y]`. The iterative computation performed by each process is the following:

```
for(hy = 0; hy < own_y; hy++)
{
      for(hx = 0; hx < own_x; hx++)
      {
            int iteration = iteratePixel(hx+own_x*rank,
               hy);
            mydata[hx][hy] =
               determinePixelColour(iteration);
      }
}
```

Since the partitions are identical with respect to the global image, each process needs to compute the positions of the pixels in their local block with respect to the global image from the size of its dimension and its rank, which is not intuitive.

This can be avoided in MPI, if global pixel positions are initially computed on a single process, say process 0, and then scattered across the rest of the processes, but not only is this another complex communication, but also it can be only effective when the image is split in a particular dimension, horizontally or vertically, since multidimensional arrays are stored sequentially in memory. MPI can only communicate sequential bits of memory in one message without additional function calls. This discussion alone provides examples to the low-level considerations of an MPI programmer, which are in stark contrast with the accessible data distribution flexibility provided by Vidim.

What is more, by using the PGAS programming model, Vidim provides the programmer with the freedom to keep all data in one local memory and program other processes to access it directly, as demonstrated in the first implementation of the Mandelbrot Set case study shown in Figure 5.3. While this would require some explicit synchronisations of the concurrent accesses, it can be considered simpler to program than keeping track of data distribution. More importantly, Vidim makes explicit synchronisations easy to program and keep track of, as shown in Section 5.3.3.

5.3.5 Providing Comparable Execution Performance

While the focus of this dissertation is the *programmability* of the implemented parallel programming approach with *performance* being only a secondary consideration, it still needs to be considered when discussing parallel programming as it is the reason for the existence of parallel programming in the first place. In order to demonstrate how the performance of a program written in Vidim compares to that of the same program written in other parallel frameworks, execution experiments were performed with all implementations of the Mandelbrot Set case study presented in section 5.2.2. Figure 5.8 shows the execution times of all implementations. The results displayed are the average of five executions and do not include the time taken to write the image file in the end of the program.

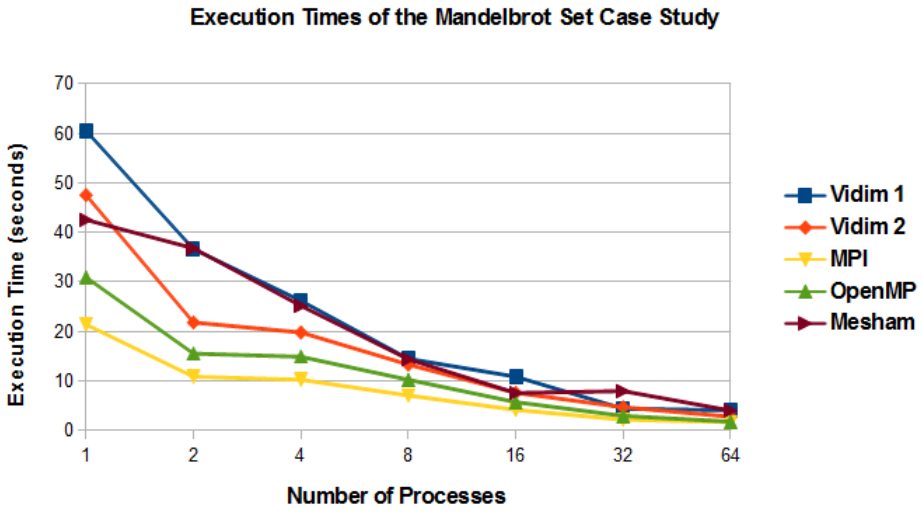


Figure 5.8: Execution times of the Mandelbrot Set case study implementations vs the number or processes.

As seen in it, MPI and OpenMP produce the best runtimes, which is not surprising, as they are lower-level programming frameworks. The results of the two Vidim implementations demonstrate that the higher-level of abstraction provided by the language does incur some performance cost, particularly for smaller numbers of processes. Nevertheless, the magnitude of this is not greater than the differences between the results produced with the textual frameworks. Therefore, it can be concluded that Vidim produces comparable performance.

5.4 Summary

The current chapter introduced two case studies, Prime Factorisation and Mandelbrot Set, and presented their implementations in Vidim. Analysis of the implementations showed that Vidim assists the programmer in the following ways:

- It clearly visualises data allocation, which helps both data distribution and communications.
- It allows direct remote memory accesses.
- It allows automatic distribution of arrays across processes.
- It clearly displays what tasks are executed by what processes.
- It provides distinguishable appearance of critical and challenging parallel programming aspects, such as explicit synchronisations and collective communications.

Furthermore, the case study implementations in Vidim were explicitly compared against the evaluation criteria identified in Chapter 2 and used to assess Vidim's programmability. The evaluation demonstrated how Vidim successfully satisfies all programmability criteria. Table 5.1 shows Vidim's results, together with those of the parallel programming solutions whose analysis was used to define the criteria.

Evaluation Criteria	**OpenMP**	**MPI**	**CODE**	**VPE**	**Mesham**	**Vidim**
General programming model	✓		✓		✓	✓
Abstraction of low-level communication details	✓				✓	✓
Clear synchronisation		partial	✓	partial	partial	✓
Clear data distribution	partial				partial	✓

Table 5.1: Comparison of parallel programming approaches with respect to usability evaluation criteria, including Vidim.

Finally, analysis of the average runtimes of all implementations of the Mandelbrot Set case study showed that performance achieved by the Vidim programs is comparable to that obtained with other parallel programming frameworks.

Chapter 6

Conclusions

The aim of this project was to design and evaluate a visual approach to parallel programming. The objective is to enable the quick and easy development of parallel code by visualising some of the traditionally challenging aspects of parallel programming.

As a result, a new visual parallel programming language, named Vidim, was defined and an end-to-end visual system to enable programming in it was developed. To evaluate the outlined approach, tow case studies, namely computation of Prime Factorisation and drawing of Mandelbrot Set, were implemented using the developed system and assessed with respect to a set of predefined evaluation criteria.

The following subsections summarise each of the three results of the dissertation, suggest possible future work and provide and overview and discussion on the project process and the lessons learnt in its course.

6.1 Vidim Definition

Vidim was defined as a new visual parallel programming language, which is based on the general PGAS parallel programming model. Its aim is to use visualisation to provide a natural representation of parallel programs. After a number of different considerations and designs, it was decided that Vidim visualises processes, together with their local memories and code areas, next to one another, and allows the programmer to individually allocate variables and specify tasks to each process. In this way the programmer can see at all times where the program's data is allocated, as well as what tasks would be executed by all processes, which aims to assist them in taking correct programming decisions.

Vidim's definition includes a set of program constructs, which are used by the programmer as building blocks of their parallel application. The language also provides a set of graphical controls, which allow the programmer to define each construct easily, by selecting the correct control options. In addition, it automatically displays the defined

aspects of the program in customised and distinguished appearance, which assists the programmer in keeping track of them.

6.2 The Implemented System

The visual system which was developed during this project enables the parallel programmer to design, implement and compile a Vidim parallel program. It was implemented in the **Java** programming language using the **Swing** GUI development toolkit. An object-oriented approach in which each parallel program construct and system component was described and implemented separately was used.

The developed system consists of a GUI, which implements Vidim's visual program constructs, and a Translation Engine, which translates the parallel program into Mehsam textual code. The programmer can then use the existing Mesham compiler to compile this code into an executable file.

The system was developed into three separate Java projects, the GUI, the Engine and the Library projects. As their names suggest, the GUI project and the Engine project implement the GUI and the Translation Engine respectively, while the Library project implements the *internal abstraction* of the parallel program used by both the GUI and the Translation Engine. The internal abstraction consists of a set of objects which describe the parallel program constructs defined in Vidim's design.

In addition, the system uses XML as an intermediate representation between the GUI and the Translation Engine. This achieves two things. From conceptual point of view, it enables the programmer to easily and compactly save their parallel programs as XML files. From software development point of view, it allows for the logical decoupling and independent development and maintenance of the GUI and the Translation Engine. The XML reader and writer were also implemented in the Library project, as they were used by both the GUI and the Engine.

The implementation of the GUI followed Vidim's language design, providing matching visual components to the parallel program's constructs. In addition, the GUI was developed to identify certain dependencies between different program components which would further assist the programmer and prevent errors.

The implementation of the Translation Engine consists of a single function, which uses the XML reader to retrieve the internal abstraction of the parallel program from the XML representation, and produces a Mehsam textual file, based on the close mapping between Vidim's program constructs and Mesham's types.

6.3 Evaluation

The evaluation process consisted of identifying and implementing two case studies in Vidim. These were Prime Factorisation and Mandelbrot Set. Vidim's flexibility allowed for two separate comparable implementations of the Mandelbrot Set case study.

Through close analysis of all Vidim implementations, the ways in which Vidim successfully simplifies parallel programming were identified. In addition, further analysis with consideration of the predefined usability evaluation criteria demonstrated that Vidim's visualisation mechanisms, as well as its use of the PGAS programming model, allow it to successfully satisfy them. Finally, brief performance analysis of the Mandelbrot Set case study showed that while Vidim's mechanisms induce some performance cost, its performance is still comparable to that of other widely used parallel programming frameworks.

6.4 Future Work

While the definition and implementation of Vidim developed in the course of this project addressed some of the most common challenges of parallel programming, there is a lot of room for future work in the conceptual design of Vidim, the current implementation of the system and the performance analysis of the proposed approach.

6.4.1 Conceptual Design

It is a well known fact that loops are one of the most common sources of parallelism in many applications. This is why they are usually paid special attention when parallel programming is discussed and mature and widely used parallel frameworks, such as OpenMP, support sophisticated techniques for their scheduling and workload distribution. Currently, Vidim does not support parallel loops explicitly, but allows the programmer to replicate the body of a loop across processes, which then execute it in parallel. This requires the manual distribution of workload in the loop. A natural further extension of the language would be introducing the concept of parallel loops, which can be specified and represented separately.

In addition, codelets is a concept unique to Vidim. Their purpose is to allow the programmer to enter particular important tasks, such as synchronisations and collective communications, separately with the help of the graphical controls provided by Vidim. They also enable the system to display these bits of code with distinctive appearance. However, the current definition of codelets means that some program's logical blocks, contained within braces, might end up split into separate codelets, as demonstrated in Figure 5.2 for example. A potential future improvement of the language could be extending the codelet concept to allow entire logical blocks to always be contained in a single codelet.

6.4.2 Current Implementation

There is a wide range of useful aspects which can be added to the implementation of a visual system like Vidim's, which simply did not fit in the 16-week span of this dissertation. Examples include

- The ability to remove added variables and functions.
- Undo & Redo functionality.
- Support for collective communications in addition to *allreduce*.
- The ability to select particular codelets from different processes and align them upon the programmer's wish. This would provide the programmer with a higher level of flexibility in the visual design of their parallel program.
- Incorporate a call to Mesham's compiler directly into the Java project on Linux machines. While the implemented system is conceptually end-to-end, the standalone Java project currently only produces Mesham code and the programmer needs to run the Mesham compiler separately to produce an executable.

6.4.3 Performance Analysis

In the current project, performance achieved by Vidim was only a secondary consideration, which is why the performance testing and analysis provided in the dissertation are very brief. However, the results obtained showed that some performance penalty is induced by Vidim. This means that a more detailed analysis to identify the particular sources of it, with carefully chosen case studies, will be appropriate in the future and could produce interesting results and discussions.

6.5 Project Discussion

The project plan and risk analysis which were identified in Semester 2, as part of Project Preparation prior to the commencement of the full-time work, can be found in Appendix D. Overall, they provided a relatively realistic prediction for the project's process and encountered problems. For example, some of the identified risks did emerge. One of them was misestimation for development and write-up time and another was occasional unexpected behaviour of Mesham. In both cases, the mitigation strategies identified proved successful.

In the early stages of the project, some unexpected deviations from the plan were caused by its change of direction. As discussed in Chapter 3, the proposed conceptual design changed early in the course of the dissertation, which required redefining some of the project's requirements. As this happened early on, there was no big penalty in terms of

time and work lost and the project stayed on track. However, this risk could have been predicted, due to the project's research-oriented nature.

Another risk which emerged during the project development was what is commonly known within Software Development as *gold plating*. It was enforced both by the deliberately vague initial requirements set for the project, which left a lot of room for invention, and by the nature of the developed visual system, which could be easily extended in many interesting ways. Weekly supervisor meetings and deadlines were the strategies which allowed for the mitigating of this unexpected risk.

In summary, some of the risks encountered during the project were predicted and planned for during the preparation stages. Nevertheless, others emerged unexpectedly and required flexibility, discussions and re-adjustment. Arguably, this is common within any project of which software development is a big part. Thus, one needs to be prepared to plan for the occurrences of unplanned hardships.

Bibliography

[Lee, 2003] LEE, P.A. and WEBBER, J. *Taxonomy for Visual Parallel Programming Languages.* University of Newcastle upon Tyne. School of Computing Science Technical Report Series 793, 2003

[Browne,1994] BROWNE, J.C., DONGARRA, J., HYDER, S.I., MOORE, K. and NEWTON, P. *Visual Programming and Parallel Computing.* 1994

[Newton, 1992] NEWTON, P. and BROWNE, J.C. *The CODE 2.0 Graphical Parallel Programming Language.* University of Texas at Austin. Proc. ACM Int. Conf. on Supercomputing, July, 1992

[Webber, 2001] WEBBER, J. and LEE, P.A. *Visual Object-Oriented Development of Parallel Applications.* Journal of Visual Languages and Computing - VLC , vol. 12, no. 2, pp. 145-161, 2001

[Beguelin, 1991] BEGUELIN, A., DONGARRA, J.J., GEIST, G.A., MANCHECK, R. and SUNDERAM, V.S. *Graphical Development Tools for Network Based Supercomputing.* Proceedings of Supercomputing 91, pp. 435-444, 1991

[Lee, 2004] LEE, P.A., HAMILTON, M.D. and PARASTATIDIS, S. *A Visual Language for Parallel Object-Oriented Programming.* University of Newcastle upon Tyne. School of Computing Science, 2004

[Newton, 1994] NEWTON, P. and DONGARRA, J. *Overview of VPE: A Visual Environment for Message-Passing Parallel Programming.* The University of Tennesse, Knoxville, Technical Report UT-CS-94-261. 1994

[Bull, 2012] BULL, M. *Threaded Programming: Course Slides, Version 1.1.* Edinburgh Parallel Computing Centre. The University of Edinburgh. 2012

[Henty, 2012] HENTY, D. *Message Passing Programming: Course Slides.* Edinburgh Parallel Computing Centre. The University of Edinburgh. 2012

[Stitt, 2010] STITT, T. *An Introduction to the Partitioned Global Address Space (PGAS) Programming Model.* `http://cnx.org/content/m20649/1.7/`. Connexions, March 16, 2010.

[Green, 1996] GREEN, T.R.G. and PETRE, M. *Usability Analysis of Visual Programming Environments: a 'cognitive dimensions' framework.* Journal of Visual Languages and Computing, vol. 7, pp. 131-174, 1996

[Brown, 2010] BROWN, N. *Type Oriented Parallel Programming.* Durham University. PhD Thesis. January, 2010

[Brown, 2013] BROWN, N. *Mesham Parallel Programming Language.* http://www.mesham.com/article/Mesham. Accessed on: July 5, 2013

[Oracle, 2013] ORACLE, *Lesson: Simple API for XML (The Java Tutorials > Java API for XML Processing (JAXP))* http://docs.oracle.com/javase/tutorial/jaxp/sax/index.html. Accessed on: June 7, 2013

[Scocco, 2013] SCOCCO, D. *The Sieve of Eratosthenes (Implemented in C).* http://www.programminglogic.com/the-sieve-of-eratosthenes-implemented-in-c/. Accessed on: August 12, 2013

[Wilson, 2013] WILSON, G.V. *The History of the Development of Parallel Computing.* http://ei.cs.vt.edu/~history/Parallel.html. Accessed on: August 16, 2013

[TOP500, 2013] *TOP500 Supercomputer Sites.* http://top500.org/. Accessed on: August 16, 2013

Appendix A

XML Structure

This appendix displays the structure of the XML representation of a Vidim parallel program.

```
<?xml version="1.0"?>
<program>

  <numproc>

  <globalVariables>
    <globalVar>
      <globalVarName>
      <value>
    </globalVar>
    .
    .
  </globalVariables>

  <functions>
    <func>
      <funcName>
      <returnType>
      <arguments>
        <arg>
          <funcArgName>
          <funcArgType>
        </arg>
        .
        .
      </arguments>
      <body>
    </func>
    .
    .
```

```
  </functions>

  <variables>
    <var>
      <name>
      <isArray>
      <elementType>
      <allocated>
        <single> or <multiple> or <distributed>
      </allocated>
    </var>
    .
    .
  </variables>

  <processes>
    <process>
      <pid>
      <code>
        <codelet>
          <codeletType>
          <codeletCode>
        </codelet>
        .
        .
      </code>
    </process>
    .
    .
  </processes>

</program>
```

Appendix B

Mandelbrot Set Auxiliary Functions

This appendix presents the implementations of the auxiliary sequential functions in the Mandelbrot Set case study in **C**. Functions `determinePixelColour()` and `createImageFile()` use the `pixel` structure defined as:

```
struct pixel
{
      int r;
      int g;
      int b;
};
```

B.1 `iteratePixel()`

```
int iteratePixel(int hyin, int hxin)
{
	double hy, hx, cx, cy, x, y, xx;
	int iteration;

	hy = hyin;
	hx = hxin;

	cx = ((((hx / HXRES) - 0.5) / MAGNIFY) * 3) - 0.7;
	cy = (((hy / HYRES) - 0.5) / MAGNIFY) * 3;

	x = 0;
	y = 0;

	for(iteration = 1; iteration <= ITERMAX; iteration++)
	{
		xx = ((x*x) - (y*y)) + cx;
		y = (2*x*y) + cy;
		x = xx;
		if((x*x) + (y*y) > 100)
		{
			return iteration;
		}
	}
	return -1;
}
```

B.2 determinePixelColour()

```
pixel determinePixelColour(int iteration)
{
      pixel singlePixel;

      if (iteration > -1) {
            singlePixel.b = (iteration * 10) + 100;
            singlePixel.r = (iteration * 3) + 50;
            singlePixel.g = (iteration * 3)+ 50;
            if (iteration > 25) {
                  singlePixel.b = 0;
                  singlePixel.r = (iteration * 10);
                  singlePixel.g = (iteration * 5);
            };
            if (singlePixel.b > 255) singlePixel.b = 255;
            if (singlePixel.r > 255) singlePixel.r = 255;
            if (singlePixel.g > 255) singlePixel.g = 255;
      } else {
            singlePixel.r = 0;
            singlePixel.g = 0;
            singlePixel.b = 0;
      };
      return singlePixel;
}
```

B.3 createImageFile()

```
void createImageFile(char* name, pixel mydata[HXRES][HYRES])
{
      FILE *fp;
      fp = fopen(name,"w");

      if ( NULL == fp )
      {
            error(1, errno, "Unable to open file %s for
               output\n", name);
      }

      fprintf(fp, "P3\n%d %d\n%d\n", HXRES, HYRES, 255);

      int i,j;
      for(i = 0; i < HXRES; i++)
      {
            for(j = 0; j < HYRES; j++)
            {
                  fprintf (fp, "%d %d %d\n", mydata[i][j].r,
                     mydata[i][j].g, mydata[i][j].b);
            }
      }

      fclose(fp);
}
```

Appendix C

Textual Implementations of the Case Studies

This appendix includes the implementations of the two case studies, **Prime Factorisation** and **Mandelbrot Set**, in **C** with **OpenMP** and **MPI**.

C.1 Prime Factorisation

For both implementations of the Prime Factorisation case study, function `getprime(int k)` returns the k^{th} prime number using the The Sieve of Eratosthenes method and is implemented separately. The implementation used was found in [Scocco, 2013].

C.1.1 OpenMP

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

#define NUM 976

int getprime(int);

void main(int argc, char* argv[])
{
      int n = NUM;

#pragma omp parallel default(none), shared(n)
{
```

```
        int s;
        int k = omp_get_thread_num();
        int size = omp_get_num_threads();
        int divisor, quotient;

        while(n > 1)
        {
                divisor = getprime(k);
                quotient = n / divisor;
                int remainder = n % divisor;
                if(remainder == 0)
                {
                        n = quotient;
                }
                else
                {
                        k = k + size;
                }

                if(quotient == n)
                {
                        printf("%d\n", divisor);
                }
        }
}
}
```

C.1.2 MPI

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>

#define NUM 976;

int getprime(int);

void main(int argc, char* argv[])
{
        int size, rank;
        MPI_Init(NULL, NULL);
        MPI_Comm commWorld = MPI_COMM_WORLD;
        MPI_Comm_size(commWorld, &size);
        MPI_Comm_rank(commWorld, &rank);
```

```
        int s;
        int n = NUM;
        int k = rank;
        int divisor, quotient;

        while(n > 1)
        {
              divisor = getprime(k);
              quotient = n / divisor;
              int remainder = n % divisor;
              if(remainder == 0)
              {
                    n = quotient;
              }
              else
              {
                    k = k + size;
              }

              MPI_Allreduce(&n, &s, 1, MPI_INT, MPI_MIN,
                 commWorld);

              if(s == n && quotient == n)
              {
                    printf("%d\n", divisor);
              }
              n = s;
        }
        MPI_Finalize();
}
```

C.2 Mandelbrot Set

In both implementations of the Mandelbrot Set case study, functions `iteratePixel()`, `determinePixelColour()` and `createImageFile()` are implemented separately as shown in Appendix B.

C.2.1 OpenMP

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
```

```
#include <error.h>
#include <omp.h>

#define HXRES 4096
#define HYRES 4096
#define MAGNIFY 1
#define ITERMAX 1000

struct pixel
{
      int r;
      int g;
      int b;
};

int iteratePixel(int, int);
struct pixel determinePixelColour(int);
void createImageFile(char*, struct pixel[HXRES][HYRES]);

struct pixel mydata[HXRES][HYRES];

void main(int argc, char** argv)
{
      double start, end;
      start = omp_get_wtime();

      int grid_size_x = HXRES;
      int grid_size_y = HYRES;
      int max_iter = ITERMAX;
      int hy, hx;

#pragma omp parallel for default(none) private(hx,hy)
   shared(mydata) schedule(static)
      for(hy = 0; hy < HYRES; hy++)
      {
            for(hx = 0; hx < HXRES; hx++)
            {
                  int iteration = iteratePixel(hy, hx);
                  mydata[hy][hx] =
                     determinePixelColour(iteration);
            }
      }

      end = omp_get_wtime();
      printf("Time = %f\n", (end-start));

      createImageFile("output.ppm", mydata);
```

```
}
```

C.2.2 MPI

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <error.h>
#include <mpi.h>
#include <stddef.h>

#define HXRES 4096
#define HYRES 4096
#define MAGNIFY 1
#define ITERMAX 1000

typedef struct pixel_s
{
      int r;
      int g;
      int b;
} pixel;

int iteratePixel(int, int);
pixel determinePixelColour(int);
void createImageFile(char*, pixel[HXRES][HYRES]);

pixel zerodata[HXRES][HYRES];

void main(int argc, char** argv)
{
      int size, rank;
      MPI_Init(NULL, NULL);
      MPI_Comm commWorld = MPI_COMM_WORLD;
      MPI_Comm_size(commWorld, &size);
      MPI_Comm_rank(commWorld, &rank);

      int grid_size_x = HXRES;
      int grid_size_y = HYRES;
      int max_iter = ITERMAX;

      //calculate own dimensions
      int own_x = grid_size_x/size;
      int own_y = grid_size_y;
```

```
//create type for struct pixel
const int nitems = 3;
int blocklengths[3] = {1,1,1};
MPI_Datatype types[3] = {MPI_INT, MPI_INT, MPI_INT};
MPI_Datatype mpi_pixel_type;
MPI_Aint offsets[3];

offsets[0] = offsetof(pixel, r);
offsets[1] = offsetof(pixel, g);
offsets[2] = offsetof(pixel, b);

MPI_Type_create_struct(nitems, blocklengths, offsets,
   types, &mpi_pixel_type);
MPI_Type_commit(&mpi_pixel_type);

double start, end;
start = MPI_Wtime();

//declare own portion of the image
pixel mydata[own_x][own_y];
int hy, hx;

//start execution
for(hy = 0; hy < own_y; hy++)
{
      for(hx = 0; hx < own_x; hx++)
      {
            int iteration = iteratePixel(hx+own_x*rank,
               hy);
            mydata[hx][hy] =
               determinePixelColour(iteration);
      }
}

MPI_Gather(mydata, own_x*own_y, mpi_pixel_type,
   zerodata, own_x*own_y, mpi_pixel_type, 0, commWorld);

end = MPI_Wtime();
printf("Time = %f\n", (end-start));

if(rank == 0)
{
      createImageFile("output.ppm", mydata);
}

MPI_Type_free(&mpi_pixel_type);
```

```
      MPI_Finalize();
}
```

Appendix D

Initial Project Plan and Risk Analysis

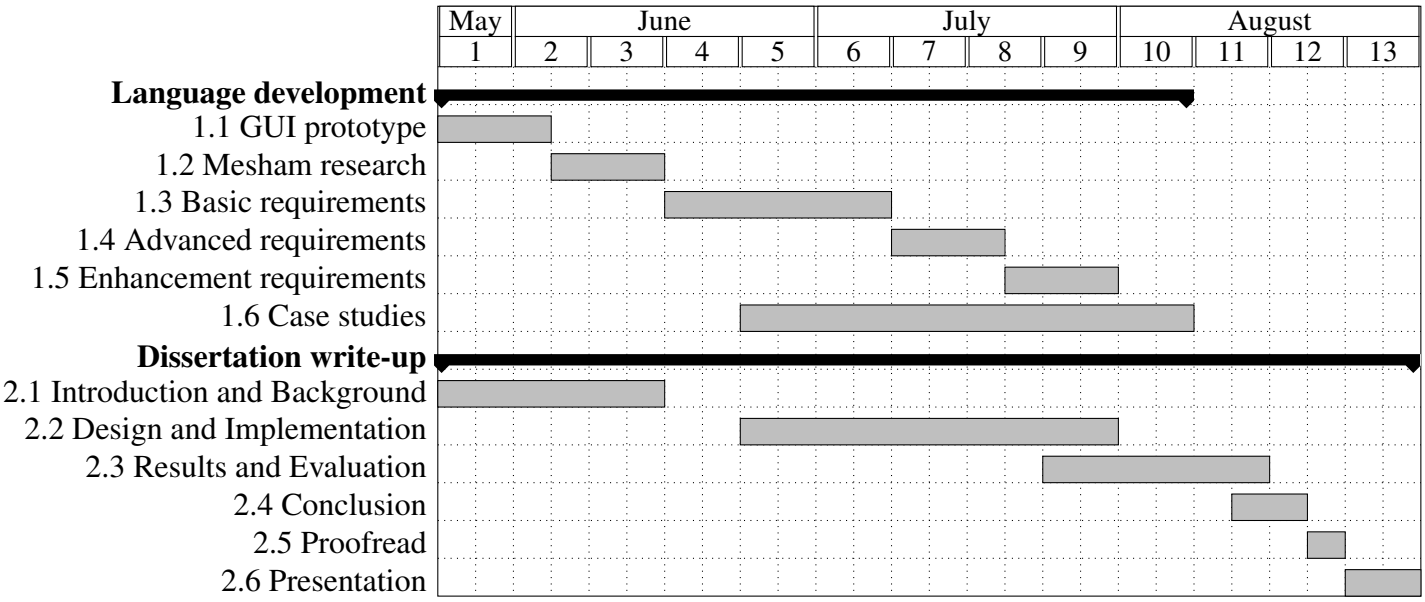


Figure D.1: Provisional Project Timeline

Risk 1: Misestimation in the development time of any of the system requirements, which might lead to failing to implement one or more of them and result in an non-functional system that cannot be evaluated.

- **Likelihood:** Medium
- **Impact:** High
- **Mitigation:** Group requirements based on priority (*Basic, Advanced and Enhancements*) and implement them one by one incrementally, in order to ensure that at least part of them will be implemented by the end of the project.

Risk 2: Lack of access to Morar at any stage of evaluation might result in delays.

- **Likelihood:** Medium
- **Impact:** Medium
- **Mitigation:** Set up a development environment on own machine.

Risk 3: Mesham is an experimental language itself, therefore unexpected behaviour of its compiler and libraries might result in delays in implementation and evaluation.

- **Likelihood:** Low
- **Impact:** High
- **Mitigation:** Schedule time for thorough investigation of Mesham. Consult Mesham's development team when issues arise.

Risk 4: Using Java Swing's drag-and-drop functionality might be insufficient for the purposes of the project, in which case additional libraries would need to be researched and used.

- **Likelihood:** Low
- **Impact:** Low
- **Mitigation:** Schedule and produce an early GUI prototype with limited exemplary functionalities in the early implementation stages in order to try out the drag-and-drop capabilities of Java Swing.

Risk 5: Hardships to evaluate the produced system, due to the research oriented nature of the project.

- **Likelihood:** Medium
- **Impact:** High
- **Mitigation:** Identify and implement case studies for each system requirement at each incremental step of implementation.

Risk 6: Time left for writing-up might be insufficient, which would lead to the failure of a timely submission.

- **Likelihood:** Medium
- **Impact:** High
- **Mitigation:** Write the first chapters of the dissertation, which include introduction and background and literature review, early on. Schedule time to write the implementation details of each project requirement and its evaluation results in the incremental steps as the project progresses.

Risk 7: Supervisor might not be available at all times during the project.

- **Likelihood:** Low
- **Impact:** Medium
- **Mitigation:** The project has a second supervisor who should be available when the first one is not.