Schedule Based Code Generation for Parallel Processors
Abstract

Dynamic model driven architecture (DMDA) is a architecture made to aid in the development of parallel computing code. This thesis is applied to an implementation of DMDA known as DMDA3 that should convert graphs of computations into efficient computation code, and it deals with the translation of Platform Specific Models (PSM) into running systems. Currently DMDA3 can generate schedules of operations but not finished code.

This thesis describes a DMDA3 module that turns a schedule of operations into a runnable program. Code was obtained from the DMDA3 schedules by reflection and a framework was built that allowed generation of low level language code from schedules. The module is written in Java and can currently generate C and Fortran code for computational tasks. Based on runtime tests for matrix multiplication algorithms the generated code is almost as fast as handwritten code.

Keywords: Parallel programming, DMDA, Dynamic Model Driven Architecture, Java, LOIS
Acknowledgements

I want to thank Prof. Welf Löve my supervisor for his support, help and guidance. I would also like to thank my parents for all their support during the completion of this project.
Content

Abstract ................................................................................................................................. i
Acknowledgements .............................................................................................................. ii
Content ................................................................................................................................ iii

1. Introduction ....................................................................................................................... 1
   1.1 Background and Motivation ......................................................................................... 1
   1.2 AIM ............................................................................................................................... 1
   1.3 Goal Criteria ............................................................................................................... 1
   1.4 Outline ........................................................................................................................ 2

2 Background ....................................................................................................................... 3
   2.1 Dynamic Model Driven Architecture (DMDA) ......................................................... 3
   2.2 LogP ............................................................................................................................. 4
   2.3 Message Passing Interface (MPI) ............................................................................... 4
   2.4 Metrics of the Code .................................................................................................... 5
      2.4.1 McCabe Cyclomatic Complexity ...................................................................... 5
      2.4.2 Coupling & Instability ..................................................................................... 5
   2.5 Code restructuring ...................................................................................................... 6

3 Design ................................................................................................................................ 7
   3.1 The prior system .......................................................................................................... 7
   3.3 Schedules ................................................................................................................... 13

4 Development ..................................................................................................................... 15
   4.1 Approach .................................................................................................................... 15
   4.2 Methodology .............................................................................................................. 15
   4.3 Initial design of the new package ............................................................................. 15

5 Implementation .................................................................................................................. 19
   5.1 CodeGeneration ........................................................................................................ 19
   5.2 Variable refactoring ................................................................................................ 19
   5.3 Implementing C ......................................................................................................... 20
   5.4 Implemeting MPI ....................................................................................................... 22
   5.5 Implementing FORTRAN ...................................................................................... 23
   5.6 Finalized design ........................................................................................................ 23

6 Benchmarks ........................................................................................................................ 25
   6.1 Metrics of the code ................................................................................................... 25
   6.2 McCabe cyclomatic complexity .............................................................................. 25
   6.3 Coupling & instability .............................................................................................. 25
   6.4 Speed of the generated code .................................................................................. 25

7 Conclusion .......................................................................................................................... 28

8 Future works ...................................................................................................................... 29

9 References .......................................................................................................................... 30

Appendix 1 Evaluation of Reflection ..................................................................................... 30
Appendix 2 Implementation of C Code for MPI. ............................................................... 32
   MatrixMultiplication .................................................................................................... 32
   ComC_MPI ..................................................................................................................... 32
   LanguageC ..................................................................................................................... 32
1. Introduction

This chapter will shortly describe the purpose and aim of this thesis project along with goal criteria and an outline of the rest of the report.

1.1 Background and Motivation

LOFAR (LOw Frequency ARray) is a sensor infrastructure developed by the Dutch ASTRON and a consortium of universities, knowledge institutes and industries. The system supports the use of very different sensors that can be used simultaneously in real time. The main application of LOFAR is in astronomy by creating an interferometric radio array of low cost radio antennas for the frequency range 10-250 mHz. The system supports other kinds of sensors that can be used for among other things agriculture and geology. The main contributors are Netherlands, Germany, UK, Sweden and France [1].

LOIS (LOFAR Outrigger in Scandinavia) is the Swedish extension of the project and aims to build 32 measuring stations across southern Sweden with Växjö as a hub. The system produces tens of Terabytes of data per second. This data can then be processed and analysed in real time if sufficient computational resources are available [2].

To handle the processing of the data a parallel computing system is required.

This creates its own problems as it is significantly harder to create programs that efficiently use parallel computing systems, compared to single computer systems.

Therefore it would be advantageous to develop tools that assist in the development of parallel programs.

DMDA3 is such a tool to aid in the development of parallel programs [3]. It is under development and can currently create a schedule of the tasks to be done for the different processors, but it does not have the capability to produce executable code. In this thesis we explore how to efficiently generate code from these schedules. Such a transformation from schedule to code must be adaptable to the specifics of the parallel computing system it is implemented for. Also, the generated code should be efficient on that particular system.

1.2 AIM

The aim of this thesis project, in order of importance is to:
• Determine how to obtain code from the DMDA3 schedules and implement this.
• Build a framework where C and FORTRAN code generated from different schedules can be executed and tested. The framework should include a number of benchmarks.
• Implement at least two benchmark programs and test the system (scheduler, code generator, runtime environment) with them.

1.3 Goal Criteria

Primary goal criteria is maintainability and usability of the framework, secondary goal criteria is efficiency of the code generating system.

Metrics for determining the success of the project regarding maintainability was:
• Modularity and structure of the implemented methods.
• Documentation of the implemented code.
• Existence and coverage of the test cases for the code.

Metrics for determining the success of the project regarding efficiency was:
• Execution time as determined by benchmark programs.
1.4 Outline

In chapter two there is a basic background of the DMDA architecture, parallel computing and code metrics. Then in chapter three DMDA3, the program that was expanded in this thesis, is described. Chapter four contains an initial outline of the methods used to expand the program, details of the initial design used in this thesis as well as design choices. Chapter five describes the development, major problems and design changes, followed by an overview of the finished program. In chapter six is benchmarks of the program for compliance with some industry standard metrics, a study of complexity and coupling as well as measure of the speed of the generated programs. Finally in chapter seven conclusions are drawn about DMDA3 and the generated programs.
2 Background

In this chapter is introduced some background concepts to allow a better understanding of the concepts used in this thesis.

2.1 Dynamic Model Driven Architecture (DMDA)

DMDA is used to transform between a service oriented conceptual view and a data driven physical level. This allows a system using DMDA to adapt to user requirement changes and requests as well as to physical changes to the infrastructure, and also to efficiently allocate computation tasks to the available processors.

In DMDA three different kinds of events are distinguished. They are user-application- and system events. User events are events triggered by user interaction. Ex. requesting that a computation is done. Application events are caused by applications in response to some predetermined condition. For example if a pattern is found in the data, the system should focus on that and send a warning. System events are caused by a signal from the system signifying a change. This could include a hardware failure.

The DMDA combines a Service Oriented Architecture (SOA) and Data-Driven Architecture (DDA) using Model Driven Architecture (MDA).

SOA views systems as a collection of services that a client can access. The Client can during runtime bind or re-bind any service or discover new services. This is good for usability and flexibility but is often computationally inefficient.

The Data Driven Architecture applies filters on a set of data that is give to the system. This allows for optimisation of the system and minimise overhead but it is hard, during runtime to change the system and writing new functions require an expert understanding of the underlying system and coding.

MDA uses step vice refinement to turn a platform independent model (PIM) into a platform specific model (PSM) and finally the PSM into a running system.

The models are special processing architectures containing components, connectors and configurations forming an abstraction of the running system. The system therefore consists of several levels of coordinators, generators and actuators that process the higher level model and generates a model of a lower level. User driven changes are handled by the top-level coordinator and cause a change in the top-level model that cascades down to the running system. The lowest level contains a probe to monitor the running system and handle system and application based events. These change the models and cascade upwards until it causes no change in the model of that level or it reaches the top level [4], [5].

This thesis is concerned with to an implementation of DMDA known as DMDA3. DMDA3 should convert graphs of computations into efficient computation code. The system begins with a editor so the user can specify the PIM and then the task graph generator creates a PSM taskgraph that is scheduled and finally turned into code (see figure 2.1). This thesis deals with the translation of PSM schedules in DMDA3 into a running system, the last step of DMDA3.
2.2 LogP

LogP is a model of multi processor computers used to analyse the performance of algorithms. The variables are latency (L), overhead (o), gap (g) and number of processors (P). Latency is the max latency for a message in the system, i.e. the time it takes for a message to reach its destination. Overhead is the computational cost to send a message. Gap is the minimum delay between messages. Processors is the number of processors in the system. These values are measured in multiples of processor cycles for the L, o and g variables. The LogP model was developed to abstractly describe a parallel environment so that one could evaluate and develop algorithms for use in such systems. Prior models often ignored the time cost of sending messages leading to very fine-grained algorithms with a high amount of inter processor communication. When such algorithms were used the communication delay could slow down the execution speed to a fraction of the estimated value. The LogP model works somewhat poorly in some special cases mostly due to not modelling hardware specific feature such as long messages, support of special routing patterns or node to node specific latency, known as distance. The distance issue is not a problem in most machines with a size range of thousands of nodes since the maximum latency difference in modern topologies is at most a factor of two [6].

LogP is the machine model currently used by DMDA3.

2.3 Message Passing Interface (MPI)

The Message passing interface (MPI) is a specification for a library interface of functions, methods or subroutines, depending on the language it is implemented in. The interface has as primary purpose to facilitate the creation of applications that use the message passing parallel programming model. It was created in the early 90’s to facilitate portability and ease of usage. Secondary goals was allowing efficient implementation of communication primitives, and allowing hardware developers to know goals on what to support.

The first specification MPI-1 was a message passing system with a predetermined amount of processes. The next version MPI-2 included functionality such as dynamic process creation, one-sided operations and parallel input and output.

The basics of MPI are communicators, processes and groups. At application start up
there is an initial group containing the processes then there a variety of tools for group management.

Once started MPI supports a variety of different methods, the most basic being sending and receiving that are used for point-to-point communication.

Initially the MPI specification provided support for C and Fortran 77, later it extended to include C++ and Fortran 90. There has been implementation of the MPI for many other languages [7].

### 2.4 Metrics of the Code

To help develop good code a large set of metrics has been produced to evaluate the quality of code. In this project the code is evaluated using metrics described below, so that it as much as possible adheres to industry standards.

#### 2.4.1 McCabe Cyclomatic Complexity

How complex the code in a module is to read and understand is related to the amount of paths on can take through the program. This is an important measurement as following through many different paths and comparing them is hard. But one can not simply count the number of paths as any program that has loops or other backwards referencing statements has potentially infinite paths. To avoid this one uses the cyclomatic number of a graph; this represents the amount of basic paths through a graph, In this case applied to a graph of the code in a module. This used as a base for calculating how complex a module is and with the decision flow as the graph(ie code), one arrives at the formula

\[
M = E + 2P - N
\]

where E is the number of edges, P the amount of connected components and N the amount of nodes in the graph.

Using this one can get a simple number that most often gives a good view of the overall complexity of a program. If a module has multiple sub-modules one can simply add their cyclomatic complexity for a total complexity number. Standard practice is that cyclomatic complexity should not be above 10 for an individual module [8].

#### 2.4.2 Coupling & Instability

Coupling describes how much one module or class is dependent on other modules or classes. High coupling often has several undesirable effects foremost that changing a single module can lead to changes in other modules. There are several different types of coupling. In this thesis work efferent coupling (Ce) and afferent coupling (Ca) is used. Efferent is the dependencies of a class and afferent coupling is the amount of dependencies on a class [9].

The Instability is a measurement of how likely and easy a class is to change and how likely it is to be effected by change. This is calculated with

\[
I = \frac{Ce}{Ce + Ca}
\]

where I is the instability, Ce the efferent coupling and Ca is afferent coupling.
This gives a value from 0 to 1 with 0 the most stable and 1 the most unstable. A low value is often good but it is often coupled with inflexibility unless the module is highly abstract and in some cases a high instability is unavoidable then the module should not be abstract. [10].

2.5 Code restructuring

When converting code from one programming language to another language or to machine code the code is analysed. The code is divided into components and those components which have similar constraints are combined into flow-based constraints and are handled with trees or graphs. This is then analysed and ordered replacing each component its equivalent in machine code or code in the other language [11].
3 Design

In this chapter is described the architecture and behaviour of the earlier DMDA3 code. This is done to facilitate the understanding of the system that this thesis is based on.

3.1 The prior system

The DMDA3 system is an application of DMDA and currently consists of three levels of modelling. These levels of abstraction are user graph, task dependency graph and tasks scheduled to processors. It is not a complete system but can handle some evaluation of scheduling algorithms. The DMDA3 manages transformation from user input to scheduled model. Prior to this thesis work, DMDA3 had no connection to an actual running system of a model and no probe to gain feedback from such a system.

The DMDA3 system has two central types of objects TaskGraphs and Schedules. It also consists of many classes to generate and handle TaskGraphs and Schedules.

The design of DMDA3 supports the assignment of machine models that describe the system they are to be implemented on to varying degrees of abstraction [3].

The current implementation uses the LogP model.

From a DMDA perspective, in DMDA3 the highest level model is the PIM, the next is the TaskGraph followed by the Schedule, with a running system at the lowest level.

The highest level consists of an Editor, a Coordinator and an Actuator. The TaskGraph level has a TaskGraph generator, Coordinator and Actuator. The Schedule level has a Scheduler Coordinator and Actuator. The lowest level contains a Compiler, Coordinator, Actuator and Probe.

The DMDA3 has only the TaskGraph Generator and Coordinator fully implemented using Grail & yEd for the Editor.

The packages in the DMDA3 are:

- interfaces
- LogPMachineModel
- taskGraphImpl
- schedules:
  - diagnosis
  - LogPScheduling
  - GlobalScheduling
- taskGraphGenerators
- testDMDA_2

Interfaces package
This package provides interfaces for the TaskGraphs and Schedules, their generators and component objects (see figure 3.1).
LogPMachineModel package.
This contains a LogP implementation of the MachineModel interface, it has both a linear and a constant implementations and a generator for both. The structure is seen in Figure 3.2.
taskGraphImpl package
This implements *TaskGraph* and the composing elements, Task and Port that are used to construct the *TaskGraphs*. This package also contains a class (Layering), for dividing a *TaskGraph* into dependency layers (see Figure 3.3).

Figure 3.3: The class diagram of the package TaskGraphImpl.

Schedules package
This package contains implementations for the Local and Global Schedule interfaces. It also contains *Node*, *ComputationNode*, *Processor*, *CommunicationsNode* and *ScheduleNode*. These are used to construct the *Schedules*. The dependencies of the classes can be seen in Figure 3.4.

Figure 3.4: The implementation and dependency diagram of the schedules package.
Diagnosis package
This package has two classes that test *Schedules* for compliance with the LogP model (see figure 3.5 below).

Figure 3.5: The classes of the diagnosis package.

LogPScheduling package:
This contains the several different scheduling algorithms for the LogP machine model. *AbstractLogPScheduler* is the base class that other scheduling algorithms use as a base. It itself is not for scheduling but it has contains many supporting methods that child classes can use (See Figure 3.6).

Figure 3.6: The class diagram for the LogPScheduling package.

GlobalScheduling package
These classes handle global scheduling. It is contains less scheduling algorithms than local scheduling (See Figure 3.7).
taskGraphGenerator package
The Classes in this package generate different types of task graphs. It can generate Diamond, FFTT, InverseFFT, Wave, SendTree and ReciveTree TaskGraphs of selectable size (see Figure 3.8). The generated TaskGraphs have one or two types of computation nodes.

Figure 3.8: The class diagram for the taskGraphGenerator package.

testDMDA_2 package
This package contains the testing units for scheduling and taskgraphs. The dependencies between the packages as found by Metrics [12] add-ons for eclipse [13], are shown below in Figure 3.9. The central package is the interfaces package on which most other packages rely on for interfaces for the classes they use. Then there are the packages: LogPMachineModel, taskGraphImpl and schedules these specify the basic components of the program. Then there are packages for working with and testing these parts. There is some degree of cross dependency between these layers.
3.2 TaskGraph

TaskGraphs are one of the core components of the DMDA3 as turning TaskGraphs into code is the objective of the DMDA3. A TaskGraph consists of a set of nodes. Each node in the TaskGraph represents a computation that is to be done (see Figure 3.10 for an example of a TaskGraph). Each node thus contains a string that specifies what computation it should do. The computation requires a set of input values and returns a set of output values to be given to other nodes in the TaskGraph. These input and output values are not necessary a single data primitive but represents data dependencies to other tasks.

Figure 3.10: Diagram of the TaskGraph for the computations (A+B)*(C-D).
The input values are not necessarily commutative as the computation for example could be a matrix multiplication and the output values may have to go to a specific recipient. Therefore one must have a way to determine what data is to be transferred to a specific location. The input and output locations are stored in the form of two arrays of Port objects. Each Port object has a size number, a pointer to the owning task and a pointer to another Port. That other Port itself refers to the recipient task. Figure 3.11 shows an overview of how two tasks are connected to each other. This two-way reference between Ports forms the connections that make the nodes into a directed graph of dependent nodes.

![Diagram of one dependency link between two tasks, where Task B is dependant on Task A.](image)

3.3 Schedules

Schedules are other core components in the DMDA3 as they are created from Tasks and they not only specify how the code shall be structured but also are used to evaluate the speed of the code to be generated.

Schedules are derived from TaskGraphs and specify on what processing node the tasks should be on and in what order. A schedule also contains communication nodes that signify communication between tasks on different processing nodes. Figure 3.12 shows an example Schedule for a simple TaskGraph for the computation \((A+B)\times(C-D)\) and how it is scheduled onto two processors. Processor 1 handles the addition of A and B and waits for data from processor 2 before multiplying. Processor 2 subtracts D from C before sending the result.

A communication node can be of send or receive type. Communication nodes have a Port that points to the communication node that handles the other end of the communication and a set of Ports that contain the nodes that use that communication node to communicate.
There is a distinction between two types of Schedules, LocalSchedules and GlobalSchedules. LocalSchedules describes a Schedule for a single set of dependant computational task, while a GlobalSchedule can contain several different tasks to be run on the final system and that might have no relation to each other beyond that they should be run in the same parallel processing environment. The system for GlobalScheduling in DMDA3 is not as developed as the system for LocalSchedules. The module developed in this thesis work focus on the use of LocalSchedules but should work as well with GlobalSchedules.

How the scheduling is done depends on the scheduling algorithm used and part of DMDA3’s goal is to be able to evaluate and compare different classes of scheduling algorithms.

There are currently several different scheduling algorithms that use different modes of communication such as all-to-all or point-to-point.

When the system is turned into a schedule a set of communication nodes are inserted that handles the communication, these are either send or receive nodes.
4 Development

In this chapter is a description of the development process, how the new functionality in DMDA3 should be designed. The major design choices are justified and in the end the design of the package, before implementation and testing is shown.

4.1 Approach

Evaluate and implement different methods storing and generating code from DMDA3 schedules. Create a design for the chosen code storing/generating method and use it as a basis for writing the required code. Design a metric testing system that measures computation span and frequency and possibly other metrics. JUnit test cases shall be created for all implemented code that is to be part of the final DMDA3 system.

4.2 Methodology

The code development used elements from extreme programming [14], and UPEDU [15]. There was an initial phase of usecase specification and preliminary design. Then preliminary implementations of the core classes were created. Using an iterative process with frequent testing the different design features were implemented one at a time. The eclipse [13] IDE was used for implementation.

4.3 Initial design of the new package

Initially it had to be considered how to obtain code from the schedules and the use cases, i.e. to turn a schedule with tasks into executable code. The description of usecases was trivial since the problem was to implement features in an internal part of the DMDA3 system. The two usecases were that the program should be able to generate a program from a schedule and it should be able to create a program that gives testing data back to it.

It was decided that the program should create functional code in a desired programming language. This was because creating a system to create functional machine code was beyond the scope of the project and existing compiler are highly optimized and very efficient at this task.

For design, primary C and then FORTRAN 77 code was to be generated. C and FORTRAN are among the most common low level programming languages and many other languages such as C++ and C# are based on them.

However, the system should be so flexible that there is a possibility to implement other languages.

The communication system protocol to be used is unknown and the system must be open.

To test the design a protocol had to chosen MPI, a point-to-point communication method was selected. This was because the protocol was well known and a MPI network existed on the University network. Also the fundamental structure of the schedules in DMDA3 was in the form of point-to-point communication.

The schedules only contained a basic string for storing computations. Many ways of using this was considered including:

- Storing the computation code in the nodes;
- Reflexive method calls with the node storing the method name;
- Expanding the node too contain extra data about the required task.

Of these, reflexive method calls was the method chosen, primarily because scheduling of the nodes takes significant processor time and requires manipulation of the nodes, therefore making the node objects larger by storing the computation code or task data in
the node or could increase runtime.

Reflexive method calls was found to be less than two orders of magnitude more expensive than a normal method call (see Appendix 1). Saving the results of the reflexive method calls and reusing them when possible reduced the cost to a linear function of the different types of operations to be done.

There was a need to make so that the generated code did not have too much duplication of code. Duplication leads to larger program size and is harder to optimise. Therefore it was decided that the schedule dependent code should be function or subroutine calls. Then it was necessary to make sure that the function definitions were included if needed and not duplicated. This extra code is henceforth termed support code. To manage this, the supporting code was stored in a hashmap with the functions and unique names as key. This allowed the inclusion of support code if necessary and since hashmap is a set there can be no duplications.

The different tasks have data dependencies; tasks require that the results of previous tasks are available to them. There are many ways to accomplish this. The variables and destination can be assigned in the graphing stage, in the scheduling stage or it can be inherent in the function being dependant on the graphing stage. The variables could be given names that correspond to origin and destination as both origin and destination can find this information. The only remaining problem would then be to handle communication, this could be accomplished by having a set of lists with variables that should be moved and having the communication nodes reading from these lists. Another way would be to have a set of lists for destination and make the linked destination node use the same variable as output. The outvariables could refactor the invariable of the destination.

The last method (variable refactoring), was chosen as the method to be used. This method efficiently handles intra-processor variable transmission. When inter-processor communications occur, the target node has an outvariable that is the same as the invariable of the sending node. The outvariable will also be refactored this will propagate latter but since the computations happens in dependency order no extra consideration has to be taken for these cases. For intra-processor communication the outvariables associated send and receive tasks must be found and the variable name changed appropriately.

Implementing this requires that function or subroutines outvalues be the same as the invalues of the functions that depend on them. This is achieved after all code parts have been listed the system goes trough each task, changes the outvalues so they have unique names and then change the invalues of all depending tasks to the same.

To handle intra-processor communication an interface was designed. The class it specifies should handle communication and general language code such as headers, start & end of the main function etc. The communication generators should be written by the advanced user to fit to the specific computer system, in this thesis MPI was implemented.

To be compileable code requires a specific set up. This includes program start and end. This design assigns this to a language specific module that takes the main code string and support code and wraps it in start up and end functions and also places the support code in the correct locations.

The design created for this thesis only required the addition of two new packages. codegenrator containing the classes that generate the code and languageimpl containing the language specific implementations (see Figure 4.1 for insertion points).
The codegenrator package was designed with 4 classes and 2 interfaces (see Figure 4.2). CodeCombiner processes a schedule into source code. CodeContainer and Variable Container store code for tasks respective variables. LanguageCodeGenerator and CommunicationCodeGenerator were interfaces for specific programming languages and different communication types. CompileAndRunCode tries to compiled and run the source code.
Figure 4.2: A simplified version of the initial design of the codegenrator package. This is not the final design. It contains two interfaces and four classes.

Summing up the design created (see figure 4.2), it was built around a CodeCombiner class that processed a schedule and created lists of CodeContainers that were turned into a set of code files. The CodeContainer and VariableContainer were simple classes that were only used to store information about the classes. There were two interfaces ComunicationCodeGenerator and LanguageCodeGenerator. The first handles the specifics of the communication, the second the target programming language. Finally there was a compile and run class (CompileAndRunCode) only to simplify the compilation and deployment of the generated code.
5 Implementation

In this chapter the implementation of the program is detailed. It begins with the overall development and then show the specific problems found when implementing C, MPI and FORTRAN. The chapter ends with a description of the final design of the program.

5.1 CodeGeneration

Since the system relies on existing compilers, function calls can be used as the compilers will be using inlining where it is efficient and a great deal of work has been done to make exciting compilers efficient. Therefore the main code can have a list of function calls (determined by the Schedule), and have the different functions written in, therefore It was decided to in the CodeContainer separate main code and support code. Support code is to be written once per appearance in the schedule and main code to be written multiple times in the order described by the Schedule.

The user of DMDA3 would then only be required to write the functions for their desired computations, which is relatively straight forward. The system is also relatively simple to expand as the SubCodeGenerators can be replaced with other methods as long as they return a CodeContainer object.

There is a CompileAndRunCode class that contains methods to attempt to compile and run the program. The CompileAndRunCode class uses calls to implementations of the LanguageCodeGenerator that contains command line code for compiling and executing the code on the machine. These commands are machine specific as how one compiles and executes programs vary from system to system. CompileAndRunCode gives the LanguageCodeGenerator a string containing the name of the current operating system. This gives the possibility of having a relatively generic system if the LanguageCodeGenerator uses the string. If this is not possible, one must simply compile & run the code manually. For evaluating the generated code there is a test option that inserts timers at critical locations. These timers return the recorded values after code execution.

When implementing a new program using the codegenrator package there are four levels of potential complexity:

The most basic level is if the program does computations that have already been implemented then only the task graph has to be created.

Next, if it is an implemented language and using a preimplemented communication protocol, then only the code for the methods, in string format, has to be filled in and to create the TaskGraph.

The third level is if one uses an unimplemented communications system. Then one must also implement a ComComGenerator to generate the code for communication. This is harder and requires knowledge of the communication system and some knowledge about DMDA3.

The most complex level is if it is a new programming language. Then one also has to implement a LanguageCodeGenerator. This is about as complex as implementing the communication module but requires more knowledge about DMDA3 and the target language.

5.2 Variable refactoring

Data must be sent between the tasks in the program. The Schedule has a list of Ports determining where to send the data to and from a given tasks. The Ports only specifies that data should be sent there and the size of the transferred data. This leads to the code generated having to handle in and out variables from the functions and subroutines. The
communication order is deterministic, as the ports are in an ordered list; therefore one could use the order of communication as a method of determining the destination of the data. But this is hard to see from the perspective of the DM2DA3 user which only writes the computation code.

The only limitation this has is how to handle multiple outputs as functions normally only allow one return value. This can be solved in C by pointer arguments and in FORTRAN by subroutines. In general most programming languages allow for returning data via the arguments. This is relatively simple to code but not as simple as a return statement. Determining which variables to send and which are to be received is not so simple. One method is to use the Tasks in and out degree. If so, the in degree first input variables counted from of the function would be input values. The out degree last variables would be out variables, there could even be overlap of the two. This does require thought and structure that is not common in coding and was therefore not implemented. Instead two ArrayLists were used one for each communication direction with the two ArrayLists having the same order as the ports.

The variables are stored as VariableContainers in the CodeContainer object where they are used.

With this structure DM2DA3 can refactor the variables, this is done so that for each node it is given unique variable names for its out variables and the code in the function is refactor to account for this. To get unique names for variables the unique id number of the Task and the number of the Port it were combined and added to its earlier name. Then all Ports find their corresponding Task and refactor the invariables there to that name. To correctly instantiate each variable, the variable object also stores an extra string containing code that contains the information required to instantiate the variable.

A problem found was that not all programming languages handle freeing unused variables automatically which could use up the available memory.

To solve this each variables was given a counter per processing node that recorded how many times it was used and when the code was combined the system counted down until it had been used a number of time equal to the time it has nodes associated with it so that afterwards it could call the LanguageCodeGenerator and it could free the variable if necessary.

This was later found to be even more complex as the variables could be data structures that might not be freed correctly by any standard function but instead required a specially written release function. Therefore all code for a task should have support code for freeing any unusual data type if necessary for that language. The name of such a release function should be such that the language generator for that coding language could merely from the variable object call that release function. For C this was implemented as each release function should have the name “variable type+Free” so that the language generator can simply put that string when a variable can be freed.

5.3 Implementing C

To test and develop the system it was necessary to implement a coding language and a TaskGraph with computations to be done.

An overview of the sequence of events that occur when the program creates code for one processor from a schedule is shown in Figure 5.1. The program first gets the code for all tasks, it does so by checking if it has a example of the code for such a Task, if not it uses reflection to get it otherwise it makes a copy of a prior version. Then it refactors the variables for all nodes. All CodeContainers are combined and then it uses the LanguageCodeGenerator for that programming language to create the code that it saves to a file.
The first implementation was the programming language C, and matrix multiplication as a task. Matrix multiplication was chosen since it is a common operation and there was prior experience in implementing it on parallel computing systems in C.

First a LanguageCodeGenerator was implemented for the language. It had as input a string with the main code, a set of support code strings and the number of the processor it was supposed to generate the file for.

The first implementation tried of this simply wrapped main code for that processor in a main method and put the support code above it. Then two subclasses of CodeContainer were created, one for generating a matrix and one for multiplying two matrices. Both had a main code string containing a function call and a function as the support code.

This did not work for several reasons.

Variables in C must be instantiated before they are used and should not be instantiated more than once. This was solved by having the variables as input for the LanguageCodeGenerator so that it could instantiate them at the beginning of the code.

The code required imports and the imports should be first in the code. To solve this several solutions were tried including creating several types of support code. In the end it was chosen to store all support code as a tuple containing the code and an integer that decided where in the code that specific code should be placed. From studying the layout
of C and FORTRAN code and general experience with code as most code is structurally
derived from C. I chose to use six locations to place code:

- Initial: imports.
- Defining: structs and definitions used by other methods.
- Used: to place code for methods that are to be used.
- Setup: located after the start of the main method and used to setup the
  preconditions and in the languages that require initial variable initiation that is
  placed here.
- Ending: code stopping setup initiations and finalizing. This is after the main code
  but before the main method ends.
- Post: located after the main code. Code for methods that are pre declared can be
  put here.

These six locations are currently sufficient to organise the code used. To make cleaner
modules it can be recommended that one has a set of include statements and use file
linkage to create the code. This makes it possible to create and test most of the code in a
standard IDE(Integrated Development Environment) and only create modules that use
the liked file for the actual program to be scheduled.

After these changes the system could generate code for simple matrix multiplication.
The strassen multiplication algorithm was then implemented as it is a more complex
algorithm with multiple in and out values of different size for the tasks. This was
relatively straightforward to implement, with only some minor problems. The optimal
scheduler that was used to create the schedules has an exponential time complexity so
for more complex TaskGraphs it could not be used. Also there was a problem with
generating unique variable names if there were a large amount of variables. This was
solved by making the renaming algorithm more complex.

5.4 Implementing MPI

When the system could create single processor code, implementation of multi processor
code using MPI was done.

This required the implementation of a ComCodeGenerator class that took the
communications node and created a CodeContainer object for it. This process
uncovered many problems with the initial design and required many changes to the
program.

First when trying to use the system to implement MPI communication it was found
that any variable sent had to be transformed to be sent either into an array of a type
recognised by the MPI standard or to create a new MPI type for each new type of
variable. The implementation of matrix multiplication used a struct containing a two-
dimensional array of a size that varied, and integers recording the number of rows and
columns. Therefore one could either create an MPI type for the variable for each size
that was to be transferred from only the knowledge of it in the code, or transform it into
an array. Transform it into a char array was chosen, since this required less overhead
and char is the smallest data type and therefore the most precise. Then the problem
became how to move the information from the struct to the array. First methods like
typecasting and memcopy were used, but when tested the data was not transferred. It
was found that due to the usage of pointers in the struct the communication did not
work. Some data was not transmitted only the pointers to the data were sent. When the
pointers were transferred to another machine this caused errors as the transferred
pointers refereed to location that were not used on that machine or used for other
purposes.
The second solution tried was to create a function that would analyze the in variable and depending on type and pointer, transcribe the information to a char array so that a counterpart function could reverse the transformation. The possibility of the in variable being a linked list or even more complex data type made implementing this without the possibility of infinite loops or other errors beyond the scope of this project, and the computation required might be so slow as to work at cross-purpose to the projects goals of using low level languages.

The final solution was to create a framework for methods that used the variable type as a part of the name that would be implemented in the same location as the data type was declared or in the tasks that was used. These methods would be specially written to convert that data type to and from a char array. The example for C was “variable type name”+toArray and “variable type name”+fromArray.

This solution increases the burden of the users of the DMDA3 and is not elegant. The users must either be limited in their use of data types or must implement two extra functions only for the purpose of complying with the DMDA3 system. This solution was only used as no other viable solution was found.

An encountered problem was that MPI communication is blocking, and the scheduling algorithms assumed non-blocking communication for optimization and layout of the schedules. The general solution to this was not within the scope of the project since it would require updating all implemented scheduling algorithms. A simple scheduling algorithm was created. The new algorithm places a task according to which processor has the least computation time with a penalty if a node has to receive data from a different processing node. The communication is placed layer wise with the processor taking turns to schedule communications. This algorithm was enough to create some testable code, but is not optimized.

Another problem encountered was that the MPI is made for a SIMD (Single instruction multiple data) system and the initial design made in this thesis project presumed a MIMD (multiple instruction multiple data) program. This was a design error and therefore it was decided that the package should be expanded so that one should be able to select if it would create a MIMD or a SIMD program. The method here only changes the form of the generated code not the TaskGraph or scheduling. For the change to effect the scheduling the difference would have to be part of the MachineModel from the beginning if the goal is to take advantage of the architecture when scheduling the tasks.

The solution was to expand the LanguageCodeGenerator interface with an isMIMD method that would determine if the language generator is made to generate one or several output files. If there are several output files there is no change from the previous design. If it is a SIMD the CodeCombiner runs as before but the retuned code is combined into a single string that after the system has run for all processors, this string is entered into the LanguageCodeGenerator a final time with a negative processor number. This creates the complete code file.

5.5 Implementing FORTRAN

After implementing C, the implementation of Fortran was similar, the structure was mostly the same, only changing C syntax to equivalent Fortran syntax. The problems primary came from getting the MPI to work and that some variable declarations had code on both sides of the variable name.
5.6 Finalized design

In the end there were some changes to the design of the code generator package, as can be expected when using an iterative design method. However the basic design is the same. Variable refactoring has been separated to its own class to reduce the complexity and size of `CodeCombiner` (see Figure 5.2).

Figure 5.2: Final class diagram of the codegenrator package. Arrows are in standard UML notation.

A tuple `PairStringInt` class was created to store support code and its desired location. Several classes have been separated into an interface and an implementation of said interface, this is to make the system easier to modify and less implementation dependent. There have also been numerous changes to the internal structure of the classes.

In addition the language specific implementations were placed in a separate package (`languageimpl`). All classes in this package are extensions of `CodeContainer` or implementations of `LanguageCodeGenerator` or `ComCodeGenerator`.

A short description of some of the C example implementations is in Appendix 2.
6 Benchmarks

In this chapter some of the metrics of the code is shown and discussed. Then runtime tests on the generated code are shown and the speed of hand written code is compared with generated code for the same task.

6.1 Metrics of the code

To ensure that the project was following best practices and that it had low coupling and high modularity, it was decided that code evaluation tools should be used.

The PMD [16] and Metrics [12] add-ons for eclipse were chosen for this purpose as they implemented features that were needed and they were also straightforward to use with the current project. Benchmarking did show that some methods were too complex and that there was code that did not follow the Java standard. The code was changed to rectify this and afterwards the code was within recommended values (see figure 6.1).

![Figure 6.1: Some metrics of the codegenerator package, divided by the recommended max value for that metric.]

6.2 McCabe cyclomatic complexity

The cyclomatic complexity of the code has a maximum value of 7 for the method generate in `CodeCombiner`. `CodeCombiner` is the largest class and does several tasks; still the cyclomatic complexity as seen if figure 6.1 above is well below the industry standard.

6.3 Coupling & instability

The main `codegenration` package has an afferent coupling of 20 and an efferent coupling of 3. Giving it an instability (I) value of 0.13. This coupled with an abstractness of 0.33 suggests a stable and relatively flexible package.

The `languageimpl` package has an afferent coupling of 3 and an efferent coupling of 16. Therefore its instability (I) is 0.842. This high value can be allowed due to its low abstractness value of 0, and is caused by the package only containing implementations of classes from the `codegenrator` package.

6.4 Speed of the generated code

To test the efficiency of the generated code it was tested on a MPI network and the time taken for different amount of computers was recorded. For these tests C code was generated. The computation that was used for testing was the strassen algorithm for
matrix multiplication. The program divided the matrixes until they contained less than 16384 ($2^{14}$) elements then it used ordinary matrix multiplication.

Matrixes sizes of 200, 300 and 400 corresponding to 40000, 90000 and 160000 matrix elements were used.

The tests were done for a single computer system to determine the speed without any data transfer and then on two, three and four computers. One can see from figure 6.2 that the time taken has a growth above linear but bellow $N^2$. This is according to prediction where ordinary matrix multiplication grows with the power of 3/2, strassen multiplication the power of $\log_2(7)/2$ and the rest of the functions are linear, putting the growth between $O(n)$ and $O(n^2)$. There was significant differences in runtime between the computers, this can be seen in figure 6.2 from the difference between maximum and average run time of the computers. This is most likely caused by uneven task loads and differences in computer speed.

![Figure 6.2: Average and maximum runtime in ms for matrix multiplication using generated code compared to amount of data elements, using 1 to 4 computers.](image)

To get a baseline value, a program was (manually), written that did strassen matrix multiplication. As seen from figure 6.3, the manually written program was less than 15% faster in a multiple computer environment and slightly slower in a one computer case. It was slower in the single computer case for large matrix sizes, showing that the basic multiplication algorithm in the hand written code can not be significantly better than the generated code.

In the multiple computer case the speed difference is probably due to better usage of available resources by the written code, probably due to that the simple scheduling program used to create the schedule that is turned into code is not optimal. In addition the generated code schedules computation when the code is written and does not compensate for different speeds of different computers, The manually written code uses dynamic allocation to different computer. When they are done with a task they request the next task.
Figure 6.3: Comparison between the written and the generated code for one and four computers.

The relative transmission time compared to the total computation time is as shown in figure 6.4 relatively low. The jump when going from 40000 to 90000 elements is caused by there being an additional matrix division increasing the amount of transmissions.

Figure 6.4: Communication time as a percentage of the programs total runtime.

The tests were run on the university MPI network. The computers on the network were of varying speed. Also since the tests were done over a network where there is other network traffic the exact runtime varied. To evaluate this variation the program was run several times. Ten executions with 160000 elements and four processors, gave an average time of 2516.3 ms and standard deviation of 15.9 ms showing that the variation is not significant.
7 Conclusion

The DMDA was proposed to help in the writing of multiprocessors code. In this thesis project the DMDA3 program have successfully been expanded to allow for the generation of machine code.

This thesis describes the successful implementation of a method to generate code from a schedule in DMDA3, thus achieving the primary aim of the project. The implementation can be adapted to several different coding languages, and communication methods. It can be used to implement programs to do certain tasks.

The implemented method is very dependant on the schedule both for performance and more importantly, to ensure that the communication does not cause deadlock.

This implementation comes with implementations primary for C and MPI but there is also a FORTRAN and MPI implementation. If other languages and communication methods are desired one must implement modules for those. This system provides a model for how such modules should be constructed (see Appendix 2), but for different languages one might have to do some redesign. However, it has not been proven that all other languages can be sufficiently implemented using only implementation of the interfaces LanguageCodeGen and ComCodeGen.

There were problems with implementing benchmarks for the code. This was caused by the fact that the generated programs where not run inside the Java virtual machine. Instead an option to make the generated code report the time taken was made in the generated programs. This is suboptimal but provides some ability to evaluate the code.
8 Future works

There are several directions one can further develop the DMDA3 Program:

- One could expand the schedulers. More efficient scheduling or a system to select communications method independently of the task scheduling would be useful as this would allow easier and more efficient adaptation to different computing systems. Expanding the set of communication types for Schedules e.g. broadcasting, would also open the possibility of faster generated programs.

- The DMDA3 source code could be further documented and the code made to follow Java development standards more closely.

- One could also improve and expand on the language and communication code generators.

- Either to be more efficient or to cover a wider variety of languages and protocols, a multithread version is a starter idea.

- Finally a system to extract and create computation code containers from prior code could be useful. If one also could create the probe element of the system and implement feedback from the machine model into the system, all levels of the DMDA architecture would be implemented.
9 References


Appendix 1 Evaluation of Reflection

A simple Reflection testing program that compares reflection to ordinary function calls. This must be run with minimum optimization. This is only an order of magnitude test.

Results:

<table>
<thead>
<tr>
<th>iterations</th>
<th>Function call (ms)</th>
<th>Reflexive function call(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>1E3</td>
<td>10</td>
<td>51</td>
</tr>
<tr>
<td>1E4</td>
<td>30</td>
<td>160</td>
</tr>
<tr>
<td>1E5</td>
<td>80</td>
<td>1312</td>
</tr>
<tr>
<td>1E6</td>
<td>691</td>
<td>13669</td>
</tr>
</tbody>
</table>

The code:

```java
package testCodeGen;
import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
public class ReflectionTest1 {
    int lSum;
    public static void main(String[] args) throws
        IllegalArgumentException, InstantiationException, IllegalAccessException,
        InvocationTargetException, ClassNotFoundException, SecurityException,
        NoSuchMethodException{
        int k=100;
        while (k<10000000){
            long T=System.currentTimeMillis();
            for(Integer i=0;i<k;i++){
                ReflectionTest1 R=new ReflectionTest1(i);
                R.funk(i+++);
            }
            System.out.print("call test "+k++ "+(System.currentTimeMillis()-T)=" ms/n");
            k=k*10; //test again at x10 size
        }
        k=100;
        while (k<10000000){
            long T=System.currentTimeMillis();
            for(Integer i=0;i<k;i++){
                Class<?> cl = Class.forName("testCodeGen.ReflectionTest1");
                Constructor constructor = cl.getConstructor(new Class[]{int.class});
                Object invoker = constructor.newInstance(new Object[]{i});
                Method method = cl.getMethod( "funk",String.class );
                method.invoke(invoker, ""+i);
            }
            System.out.print("reflect test"+k++ "+(System.currentTimeMillis()-T)=" ms/n");
            k=k*10; //test again at x10 size
        }
        public ReflectionTest1(int i){ lSum=i; }
        public void funk(String i){ lSum += i.length(); }
    }
}
Appendix 2 Implementation of C Code for MPI.

Reference for further implementations of code.

MatrixMultiplication
This is a child class of CodeContainer with a Constructor that creates the Code for Multiplying two matrices. The new Constructor has a Task as invariable, this is the Task object that requested this object as its computation. All Computation objects should have this in it’s constructor as the system will try to give it during object creation. Then it uses the super class constructor to declare its name, maincode string and set up its containers. Thereafter variables are declared the have the same name as the variables in the main code. The two support code pieces are declared. In import call and the function for multiplying two matrices. The import call is to a file that has functions for handle matrices for the DMDA3 system. The import call is placed at location 0 as it should be first and the function is placed at location 2 just before the main method starts.

ComC_MPI
This implementation of ComCodeGenerator creates C code for communicating using C. First it creates general support code, imports at location 0, MPI start at location 3, and MPI end at location 4 after the main code. Then it checks the in nodes type to see if it is a send or receive node. It then defines a toArray / fromArray function to combine multiple sent variables to a single array. Then maincode that adds in/out variables to on array and sends/receives the variable array is written.

LanguageC
This is made for generating SIMD C code. It is tested with MPI. If this is code for a single processor it extracts any variables and support code. The support code is ordered and combined. Then it combines the maincode to a string and wraps it in if statements specifying what processor to run for and adds timing if the code shall be tested. Then it returns the maincode string.

If it gets a negative processor number it creates the string for the variable initiation. Then it combines the strings for the support code, the variables and the main code in order. It returns a string containing the finished source code.