Parallel Abstract Datatypes for Run-Time Fusion Optimisation

MEng Computing Final Year Project Report

by Thomas L Hansen,
Department of Computing,
Imperial College of Science, Technology and Medicine

June, 1999
Submitted in partial fulfilment of the requirements for the Master of Engineering Degree in Computing (Software Engineering) of the University of London and for the Diploma of Imperial College of Science, Technology and Medicine.
Abstract

Writing programs for parallel computers can be a complex and time consuming task. On distributed memory machines, programmers need to explicitly insert communication and synchronisation calls into their code in order to make processors cooperate. A particular case which has potential of generating large communication overheads is when all processors need to see the same value for certain variables at all times. As changes to these shared variables by any processor must be instantly observed by all other processors, programmers need to provide synchronisation calls for the variables throughout their code in order to ensure that a global value is maintained. This project has created a library which implements shared variables. The library has been prototyped on double precision floating point numbers. Using the library eliminates any need for explicit communication, making the code easier to read and understand. Lazy evaluation techniques are used in order to perform run-time optimisation of the communication calls to save overheads. The library has achieved very good performance results both on small test cases and on a large-scale plankton simulation. The performance observed is close to that which is achieved by performing manual communication optimisation on the code.
Acknowledgements

I would like to thank my supervisor, Dr. Tony Field, for all his wisdom, brilliant advice, enthusiasm and support during the project.

Also, thanks to Sanjay Budhia, Andy Cheadle, Öyvind Eikeland, Dave Gardner, Chris Harris and Anthony Pereira in the MEng 4 group for good technical advice, clever comments, a great sense of humour and support throughout the final year.
Contents

Abstract iii
Acknowledgements v

1 Introduction 1
  1.1 Parallel computing ........................................ 1
  1.2 Sharing variables on a parallel computer ................. 3
  1.3 Objectives ............................................. 4
  1.4 Report structure ....................................... 4

2 Background 5
  2.1 Parallel computing ....................................... 5
      2.1.1 Homogeneous vs. heterogeneous architectures ....... 6
      2.1.2 Shared memory architectures .......................... 7
      2.1.3 Message-passing architectures ....................... 8
  2.2 Parallel programs ....................................... 10
      2.2.1 Writing parallel programs ........................... 10
      2.2.2 Performance of parallel programs ................. 11
  2.3 Shared variables on message-passing architectures .... 12
      2.3.1 Avoiding explicit communication calls ............. 13
      2.3.2 Fusing communication of variables ................ 14
      2.3.3 Related work ..................................... 15

3 An ADT for run-time fusion optimisation 17
  3.1 Assumptions ........................................... 17
  3.2 Semantics of shared variables ........................... 18
      3.2.1 Operations where the result is a shared variable .. 19
      3.2.2 Operations where the result is a local variable .. 20
  3.3 Delaying communication ................................. 21
  3.4 Forcing communication ................................. 22

4 Implementation of a Communication Fusion Library 23
  4.1 Choice of implementation language ...................... 23
  4.2 Issues on parallel libraries ........................... 24
4.3  The shared double ........................................... 25
  4.3.1  The CFL_Double class ................................... 25
  4.3.2  Implementation of operations ............................ 28
4.4  The communication manager .............................. 31
  4.4.1  Delaying communication ................................. 32
  4.4.2  Forcing communication ................................. 32

5  Evaluation .................................................. 35
  5.1  The Fujitsu AP3000 parallel computer ..................... 35
  5.2  A test harness for the CFL ............................... 36
  5.3  Evaluation of small examples ............................. 37
  5.4  Case study: Virtual Plankton Ecology .................... 45
    5.4.1  Introduction ........................................... 46
    5.4.2  The ZB Model .......................................... 47
    5.4.3  Incorporating the CFL into the ZB Model ............. 50
    5.4.4  Performance of the CFL in the ZB Model ............. 51

6  Conclusion and Further Work ............................. 53
  6.1  Conclusion ................................................ 53
  6.2  Further work ............................................... 54
    6.2.1  Further evaluation of the CFL ......................... 54
    6.2.2  Adding data types to the CFL ......................... 54
    6.2.3  Relaxing the assumptions made by the CFL ........... 54
    6.2.4  Including point-to-point messaging in the CFL ....... 55
    6.2.5  Multi-threading ...................................... 57

A  Cases covered by the test harness ...................... 61

B  Notes on using the CFL library ......................... 65
  B.1  Programming with the CFL library ....................... 65
List of Figures

1.1 Cost versus performance curve and its evolution over the decades [2] .................................................. 2

2.1 Shared memory architecture ................................................................. 7
2.2 Message-passing architecture ............................................................... 8
2.3 Illustration of how code is simplified when communication calls are removed ........................................... 13
2.4 The time required for a reduction operation increases with the amount of data transferred ........................................ 14
2.5 Illustration of how the communication of two variables can be fused .......................................................... 15

3.1 Result of operation is stored in a global double ...................................... 19
3.2 Illustration of synchronisation after increment operations ....................... 20
3.3 Result of operation is stored in a local variable ...................................... 21
3.4 Example of operation sequence with no force points ............................. 22

4.1 The CFL_Double class ............................................................... 25
4.2 The effect of operations on member variables ........................................ 27
4.3 The outcome of an operation may not be known until the type of the result is known ........................................ 30
4.4 Delaying decision on what the final result should be .............................. 31
4.5 The CFL_Manager class ............................................................... 31
4.6 The events subsequent to forcing communication .................................... 33

5.1 The AP3000 interconnection network [15] ........................................ 36
5.2 Naive code for the first example ............................................................ 37
5.3 Manually optimised code for the first example ...................................... 38
5.4 Code for the first example using the CFL library .................................... 38
5.5 Timing results for example 1 ............................................................... 38
5.6 Timing results for optimised runs in example 1 ....................................... 39
5.7 Naive code for the second example ............................................................ 40
5.8 Manually optimised code for the second example .................................... 41
5.9 Code for the second example using the CFL .......................................... 41
5.10 Timing graph for example 2 ............................................................... 42
5.11 Naïve code for the third example ........................................ 43
5.12 Manually optimised code for the third example ................ 44
5.13 Manually optimised code for the third example ............... 44
5.14 Timing graph for example 3 ............................................. 45
5.15 Speedup graph for example 3 ............................................ 46
5.16 The water column divided into biological and physical layers 47
5.17 Block and stripe allocation of the layers in the water column 48
5.18 Illustration of vertical partitioning ................................. 49
5.19 Old mechanism for maintaining global value for nitrogen .... 50
5.20 Execution time for ZB model against number of processors .. 51
5.21 Speedup for ZB model .................................................... 52

6.1 Point to point messages being sent between two processors .. 56
6.2 Optimised point to point messages being sent between two processors ......................................................... 56
List of Tables

2.1 A selection of MPI calls .................................. 10
4.1 Operations that are members of the CFL_Double class .... 29
4.2 Operations that are not members of the CFL_Double class . 29
Chapter 1

Introduction

1.1 Parallel computing

When solving a problem on a computer, one needs to formulate an algorithm that will perform a number of steps required to solve the problem, write a program that implements the algorithm and execute that program on a computer. The programmer will want their program to execute as quickly as possible in order to be able to solve large problems in a small amount of time.

A standard computer has one processor, and the speed at which the program is executed depends on the performance of the processor on which the program is executed. Since the emergence of the microprocessor in the late 1970s, its performance has increased at an average of roughly 35% per year [4]. Speaking very simplistically, this means that if your program runs at a certain speed now, in a year you should be able to purchase a processor at the same price as you paid for your current processor, and your program should run 35% faster on the new processor.

However, performance comes at a cost. Figure 1.1 is taken from [2], and shows how the performance of a processor increases with cost in the 1960s, 1970s, 1980s and the 1990s. It is interesting to note the change in the curve as time goes by. While the curve for the 1960s remains nearly linear even for quite high cost, the curve for the 1990s gives quite dramatic performance increases at a low cost, but after a short while the curve flattens out and very little performance is gained even by large increases in cost.

As it gets costly to get higher performance out of a single processor, one might simply purchase many processors, each at a relatively low cost, connect them together, and a parallel computer is born. In theory, if this computer consists of $p$ processors, it should be able to perform the same computation $p$ times faster than a processor that only consists of a single processor. The problem of getting more performance out of computers at a reasonable cost appears to be solved.
Unfortunately, life is not that easy. When building a parallel computer, performance is lost in several areas. The processors will need to talk to each other in order to solve a task together. This adds an overhead to the computation time which is not present on a single processor, thus decreasing the performance of the parallel computer. Also, one will have to find a way of solving the problem which is suitable to be run in parallel. There is no use in having 1000 processors if only one can do useful work at any time. There is also a limit to how much performance can be gained by parallelising a problem, given by Amdahl’s Law [4]:

\[
\text{Speedup} = \frac{\text{Execution time for entire task without using the enhancement}}{\text{Execution time for entire task using the enhancement when possible}}
\]

(1.1)

It is important to notice that in the denominator of 1.1, the program is timed using enhancements when possible. This means that there is a limit to the speedup that can be observed when running a program on a parallel computer, and this limit is given by the proportion of the program that cannot be parallelised. If, for example, 10% of a program could not be parallelised, then the limit of the speedup that could be observed would be 10. This is expressed by the law of diminishing returns [4]:

The incremental improvement in speedup gained by an additional improvement in the performance of just a proportion of the computation diminishes as improvements are added.
1.2 Sharing variables on a parallel computer

In spite of these difficulties, parallel computers are being used in a wide variety of areas throughout the realm of computing. With their ability to perform a large number of computations in very short time, parallel computers can predict the weather, model the evolution of galaxies, predict changes in the stock market and a whole range of other tasks with, perhaps, surprisingly good performance.

1.2 Sharing variables on a parallel computer

Without the ability to communicate with each other, the processors in a parallel computer would have no way to cooperate. It is therefore essential that the processors are able to share information when it is required by a parallel algorithm. This is commonly done through one of two techniques: shared memory or message passing (see section 2.1).

It is common for programmers to parallelise algorithms by partitioning the data across the processors and running the same program on each processor. Each processor will be allocated its part of the data on which to operate. It is normal for processors to require knowledge of data that may belong to another processor, and this must be handled by the parallel program. For example, if the parallel program adds up numbers, then each processor will calculate the local sum of its data, and then a single processor needs to calculate the sum of all the local sums to get the overall value (a more efficient way of doing parallel addition is described in [2]).

Occasionally there will exist variables in parallel programs that every processor executing the program needs to know and operate on. These variables will be referred to as global or shared variables. An example where global variables are used is the case study described in section 5.4. The program in the case study models a water column that moves through the sea, and the behaviour of the plankton that live in the water column. The data in the program is distributed across the processors in such a way that every processor needs up-to-date information about the parameters of the water column, for example the total amount of biomass in the water column. This means that every time the biomass is changed by a processor, it must be sure that the other processors are informed of the change so that they do not perform calculations on invalid data. On some parallel architectures this means that the programmer needs to insert code to make sure that all processors see the correct value of the global variables at any time. If many global variables exist, the overhead associated with ensuring that they are synchronised across all processors can be extremely large, causing the performance of the parallel algorithm to drop. Also, the extra code that needs to be added for synchronisation purposes will make the parallel program more unreadable, distracting the reader from the real purpose of the program. In [5], the process of manually enhancing the performance of the
code in the case study is described.

1.3 Objectives

This project aims to design and implement a parallel abstract data type (ADT) that can handle the problem of global variables in an efficient and elegant manner. Programmers using the library should be able to declare a variable as being global, and any changes to the variable on any processor should be visible on all other processors. One of the main objectives is to make any communication required transparent to the programmer, to allow them to focus on solving the problem without concern for parallelisation issues such as synchronisation.

The ADT is evaluated using both small example programs and a real-life case study to investigate its performance impact on parallel programs. Full details of the results produced are given in chapter 5.

1.4 Report structure

The remainder of the report is structured as follows. Chapter 2 gives technical background on parallel architectures and models of parallel computing. Chapter 3 describes the design choices that were made when planning the ADT, and chapter 4 discusses the implementation of the ADT. Chapter 5 details the tests cases and the programs used for evaluating the ADT, and finally chapter 6 gives the conclusion for the project and suggests further work that can be done to enhance the ADT.
Chapter 2

Background

2.1 Parallel computing

A parallel computer can be built in a number of different ways, depending, among other things, on which control mechanism is selected for the computer. A sequential computer conforms to the single instruction stream, single data stream (SISD) control mechanism, as it involves a single processor executing a single instruction stream operating on a single data stream. For parallel computers, the following control mechanisms exist, following the taxonomy of [2]:

**Single instruction stream, multiple data stream** (SIMD). The SIMD model consists of $p$ processors that are issued instructions from a single control unit, but each processor operates on its own local data. It is important to notice that in each time step, the same instruction is executed on each processor. As a consequence, there may be idle processors in the computer while other processors are executing. An example is a conditional statement where some processors may evaluate the condition to enter the branch to be true, while others may evaluate it to be false [2].

**Multiple instruction stream, multiple data stream** (MIMD). In this model, each processor has its own local control unit and executes its own set of instructions. Different instructions may be issued to different processors in the same time step, and each processor operates on its own local data.

**Single program, multiple data stream** (SPMD). This model is a special case of the MIMD model, where the same program is run on each processor. However, as opposed to the SIMD model, SPMD does not require the processors to remain synchronised throughout the execution of the program. Each processor is free to follow any path through
the program, dictated by its local data. It is common for the programmer to insert synchronisation calls at certain points in the code to ensure that the processors that reach this point first will stop and wait for processors that are behind to catch up.

**Multiple programs, multiple data stream** (MPMD) is also a special case of the MIMD model. In the MPMD model each processor may execute its own local program on its own local data, i.e. a different program may be executed on each processor. Although the programs may be totally different, they will still need to communicate to be able to perform the overall task they have been built to perform, so they will still need to follow some common standards, e.g. for communication protocols and data formats.

With these models in mind, a discussion of the main architectures chosen for parallel computers will now follow. The discussion will focus on the types of processors and the memory organisation selected for a parallel computer.

### 2.1.1 Homogeneous vs. heterogeneous architectures

An important issue in a parallel computer is the type of processors chosen to build the computer. A parallel computer in which all the processors are identical is said to have a *homogeneous* architecture, while one that consists of different types of processors has a *heterogeneous* architecture.

The advantage of homogeneous architectures is that each processor will represent basic data types in the same way, and this means that the processors are able to exchange data without any overhead associated with converting it from one format to another. Also, the processors will be running at the same clock speed, using the same average number of cycles per instruction (CPI), and this makes the job of balancing the workload between the processors easier. Without proper load balancing, some processors risk becoming idle while others may end up doing all the work, resulting in poor parallel performance. The main disadvantage of homogeneous architectures is that the processors will tend to be general purpose processors that will perform reasonably well at a wide range of programs. By using special purpose processors (e.g. a processor specifically designed for processing graphics), the speed at which particular tasks are performed can be increased considerably.

The advantage of heterogeneous architectures is, as mentioned above, the fact that various special purpose processors can be included in the parallel computer in addition to general purpose processors. When running applications that are able to take advantage of the capabilities of the special purpose processors, the performance of a heterogeneous architecture can be better than a homogeneous architecture. However, there will be an overhead associated with converting data from the format used by one type of
2.1 Parallel computing

![Diagram of a shared memory architecture](image)

**Figure 2.1: Shared memory architecture**

processor to another, and effective load balancing will become more difficult to perform.

### 2.1.2 Shared memory architectures

Memory organisation is another important issue to consider in a parallel computer. In *shared memory* architectures all the processors see the same address space, and are able to access any part of it. A conceptual model of a shared memory architecture is shown in figure 2.1. As shown in the figure, the main memory may be organised into one large block, and each processor accesses the memory via a common data bus. With this type of organisation, each memory access takes the same time for every processor, known as *Uniform Memory Access* (UMA). As the memory is shared between all the processors in the parallel computer, any changes that are written to memory by one processor will be instantly visible to all the other processors in the computer.

Only one processor at a time will have access to the data bus, hence it is quite likely that there will be congestion, causing some processors to have to wait for access to the bus. To alleviate this problem, each processor has a local cache that works analogously to the cache in a single-processor computer (see [4] for an explanation of how a cache is organised). Having a local cache means that when a processor writes a value to memory, only its local cache is instantly updated; the main memory will be updated later. This causes serious problems with memory consistency, and extensive re-
search has gone into finding efficient ways of dealing with this problem. The solutions, however, are beyond the scope of this report.

It is not necessary for the memory to be physically organised in a single block. A very common shared memory architecture is one in which each processor has its own local memory, but this local memory forms part of a single, global address space and can be accessed by any processor. This architecture will not have UMA, because access to local memory will be a lot more efficient than access to memory physically located on a different processor.

### 2.1.3 Message-passing architectures

In message-passing or distributed memory architectures, each processor has its own local memory and its own address space. A conceptual model of a message-passing architecture is shown in figure 2.2. The only way for a processor to access data stored on another processor is by sending a message to that processor requesting a particular piece of data. The processor receiving the request will then have to find the requested data and send it back to the requesting processor.

The main advantage of a message-passing architecture over a shared memory architecture is that the programmer is able to explicitly associate data with particular processors. Each processor is able to function as a single processor computer with its own address space, cache and local store, and can take advantage of the research and development that has been put into fast memory hierarchies and efficient processing on these architectures [1].
2.1 Parallel computing

Also, as the only way that a processor can access local data on another processor is via message passing, the problem of memory inconsistencies due to accidental overwriting of memory by a remote processor does not arise.

There are disadvantages associated with message-passing architectures. The most obvious is the fact that a processor does not have direct access to the global address space, but needs to request access from other processors. In doing so, the processor slows down the remote processor it requests data from because the request must be handled explicitly by it. This consumes time that could be spent doing useful computation instead. Also, the requests for data from remote processors have to be made explicitly in the code for the parallel program, making it more difficult to write, as special knowledge of the system calls to send and receive data is required.

Synchronous message-passing

Synchronous message passing is also known as blocking message passing. When a processor sends a message using synchronous message-passing, it will block and not execute any more instructions until the message has been received by the intended receiver. Also, when a processor tries to receive a message from a source that has not yet sent a message, the processor will block and wait until the source sends the message before proceeding. A synchronous message-passing model is very easy to work with and reason about as it is guaranteed that a message will be delivered before a processor continues to execute code. However, a process may spend a substantial amount of time waiting for messages instead of doing useful work.

Asynchronous message-passing

Asynchronous message passing, or non-blocking message passing enables a processor to proceed executing its code after sending a message, without knowing whether it has been received or not. Also, a processor has the opportunity to probe its message queue, and if there are no messages in the queue, it can proceed to do useful work (it can even choose to ignore certain messages for the time being and receive them later!). There is a clear advantage in the fact that processors do not need to block while waiting for communication. However, asynchronous message-passing makes life more difficult for the programmer when they need to reason about the program and prove that it will work. Most message-passing architectures have support for both synchronous and asynchronous message-passing.

The Message Passing Interface

The Message Passing Interface (MPI) is a library designed in order to provide a standard for how parallel programs for message passing architectures
<table>
<thead>
<tr>
<th>MPI Call</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barrier</td>
<td>Stops the processor from continuing until all other processors have reached this point in the code.</td>
</tr>
<tr>
<td>Reduction</td>
<td>Perform an arithmetic or logic operation on a variable across all processors</td>
</tr>
<tr>
<td>Scatter</td>
<td>Partition data located on one processor onto all processors</td>
</tr>
<tr>
<td>Gather</td>
<td>Inverse of scatter</td>
</tr>
</tbody>
</table>

Table 2.1: A selection of MPI calls

can be written [1]. It has been designed with portability and efficiency in mind and, in theory, any parallel program written using MPI should run on any parallel computer for which an implementation of the MPI libraries exist. Although MPI consists of about 125 different functions, only 6 of these functions need to be used [1]. This means that the learning curve for programmers starting to use MPI is not steep, and as more experience is gained, more of the functionality of MPI can be utilised, making programs more efficient and elegant.

MPI contains calls for all the basic communication operations (synchronous send/receive, asynchronous send/receive) as well as a wide variety of other calls, some of which are listed in table 2.1.

2.2 Parallel programs

In the rest of this report, we will assume the SPMD model of parallel computing, running on a message-passing parallel computer using MPI for message passing. SPMD programs are characterised by the fact that they are not synchronised at all times when they run. This means that race-conditions, conditions where processors are competing for a resource and the outcome is non-deterministic [6], may arise. As race-conditions are very difficult to reproduce, it can become notoriously difficult to debug code where such conditions occur.

2.2.1 Writing parallel programs

A parallel program is often written when a sequential solution to the problem is already known. The first thing a programmer does when setting out to write a parallel program is to identify the scope for parallelism in the sequential solution. Having defined the places where parallelism can prove useful, the programmer will need to design a parallel algorithm that solves the problem by exploiting the existing opportunities for parallelism. This
algorithm must define how the problem should be solved in parallel, where the necessary inter-process communication should take place and what data each processor should operate on. The data distribution across the processors that participate in the algorithm is a very important issue that the programmer needs to consider in order to avoid uneven load balancing between processors.

2.2.2 Performance of parallel programs

Having designed and implemented a parallel program, the programmer will have to evaluate it and identify how it improves performance over the sequential version. A good way of evaluating the performance of a parallel program is to time how long it takes to execute on $p$ processors. If $T_p$ is the run-time of a parallel program on $p$ processors and $T$ is the run-time of the best known serial algorithm that solves the same problem, then the speedup, $S$, of the program is given by equation (2.1).

$$S = \frac{T}{T_p}$$

Intuitively, it would seem as if the maximum value for $S$ is $p$, indicating that each processor does exactly $\frac{1}{p}$ of the work that would otherwise be done by a single processor. However, as extra execution time will be added to the parallel algorithm by the communication and synchronisation overhead, it seems to be that $p$ represents the asymptotic limit for the value of $S$.

On the other hand, many programs exhibit speedups both equal to, and sometimes even greater (super-linear speedups) than, $p$ so the above argument is not entirely correct. One of the main reasons that super-linear speedups can be experienced has to do with memory hierarchies. When a single processor solves a large problem, it might not be able to fit all the data required to solve the problem in memory, and needs to swap from disk in order to access all the data. As reading from disk is several orders of magnitude slower than reading from memory, this will have a severe impact on the program’s performance. When the data is partitioned across many processors, a point will be reached when the portion of the data allocated to each processor will fit in that processor’s local memory. With the disk swapping overhead eliminated, the speedup experienced may be disproportionately large compared to the number of processors used.

Another useful measure of a parallel program’s performance is the efficiency, $E$, of the parallel program, given by equation (2.2).

$$E = \frac{S}{p}$$

When $S$ is less than or equal to $p$, efficiency is between zero and one. If super-linear speedup is experienced, the efficiency of a parallel program will be greater than one.
Finally, the cost of the parallel program can be calculated. This is the sum of the time spent by each individual processor on solving the problem. The cost, $C_p$, of an algorithm running on $p$ processors is given by equation (2.3).

$$C_p = pT_p$$  \hspace{1cm} (2.3)

A parallel system is said to be cost-optimal if the cost of solving a problem on a parallel computer is proportional to the execution time of the fastest-known sequential algorithm on a single processor [2].

A further measure of how useful a parallel algorithm is is the scalability of the algorithm. A parallel system is scalable if it is able to maintain a constant efficiency as the number of processors and the problem size is increased. How much the problem size must be increased when the number of processors is increased is given by the inefficiency function of the parallel program. See [2] for derivation and explanation of the inefficiency function.

### 2.3 Shared variables on message-passing architectures

As mentioned in section 1.2, there are programs in which all the processors participating in the evaluation of the program need to know the value of certain variables. These variables are said to be shared between the processors, and they can be referred to as global variables as their current value is globally known, i.e., all processors have access to their value.

On a message-passing architecture, global variables are declared just as local variables, and only the programmer knows which variables represent global values in the program. Every processor will have the opportunity to change its local copy of the global variable without the rest of the processors knowing, thus breaking the assumption that the value of the variable is globally known. In order to ensure that this does not happen, the programmer needs to write code that ensures that every single change to a global variable is made known globally, and such communication will lead to extra communication overheads. In addition to this, there will be problems determining the order in which operations have happened, e.g., one processor might increase the value of the global variable by some value, while another concurrently multiplies the value of the double by some constant. The programmer is left with the responsibility of imposing some total order on the operations and making sure that the value that each processor has stored for the global variable is, in fact, its true value. Part (a) of figure 2.3 illustrates the code resulting from changing the value of three global variables, $x$, $y$ and $z$, including the synchronisation statements necessary to ensure that the global value for these variables is maintained.

This project concerns the design of a library that will implement global variables on message-passing architectures. The library must be able to
2.3 Shared variables on message-passing architectures

\[ \text{tmp} = \text{get}_x\_\text{change}(); \]
\[ \text{sync}_\text{across}_\text{processors}(\text{tmp}); \]
\[ x = x + \text{tmp} \]
\[ \text{tmp2} = \text{get}_y\_\text{change}(); \]
\[ \text{sync}_\text{across}_\text{processors}(\text{tmp2}); \]
\[ y = y + \text{tmp2} \]
\[ \text{tmp3} = \text{get}_z\_\text{change}(); \]
\[ \text{sync}_\text{across}_\text{processors}(\text{tmp3}); \]
\[ z = z + \text{tmp3} \]

(a) (b)

Figure 2.3: Illustration of how code is simplified when communication calls are removed

keep track of all the global variables that are used in a program, and should register when changes are made to a global variable. Such changes must induce automatic communication to obtain the true value of the variable only when absolutely necessary. A programmer using this library should not need to insert explicit communication calls into a program to ensure that the value of a global variable is up to date, thus making the code for the program easier to understand and maintain. Furthermore, when a global variable needs synchronisation, the library should be able to synchronise all other global variables that need synchronisation in the same operation, saving start-up time on communication operations. The programmer should be able to use the same syntax with global variables as with local variables, making the synchronisation process that goes on behind the scenes as transparent as possible.

2.3.1 Avoiding explicit communication calls

By avoiding the need for explicit communication calls in the code, the programmer is able to concentrate on the problem that is being solved instead of worrying about maintaining a consistent global value for the variables. In part (b) of figure 2.3 it is illustrated how the code from part (a) is simplified by removing the communication calls. Part (a) and (b) of the figure are semantically equivalent, but it is easier to tell what is going on in part (b) where no explicit communication calls are made. Separating the concern of synchronising variables from what the program is doing will enable the programmer to write code more efficiently, and the code will be conceptually easier to understand. In addition, abstracting the global variables opens up several opportunities for performing automatic optimisation of the communication patterns at run-time, as described below.
2.3.2 Fusing communication of variables

On a parallel computer the overhead, \( t_c \), associated with a communication operation is given by equation 2.4 (assuming cut-through routing [2]), where \( t_s \) is a constant start-up time associated with every communication operation, \( t_h \) is the time it takes for the header of a message to move from one processor to another, \( l \) is the number of processors the header must go through to reach the destination, \( m \) is the number of words in the message and \( t_w \) is the time it takes to transfer one word.

\[
t_c = t_s + lt_h + mt_w
\]  

(2.4)

Normally \( t_h \) is small, and if \( m \) also is small (i.e. it is a short message), then \( t_s \) can have a significant impact on the value of \( t_c \). Figure 2.4 illustrates how the time required for a reduction operation varies with the amount of data involved in the operation. The data was obtained by performing 10000 \textit{MPI_Allreduce} operations on 4 processors. As the graph shows, the increase in execution time appears to be linear with the increase in data. However, it is interesting to note that as the amount of data is increased from 1 double to 300 doubles, the execution time only doubles. This illustrates how the overhead \textit{not} directly associated with data transfer (i.e. \( t_s \) and \( t_h \)) influences the overall communication time.

The reason for delaying communication is that as execution proceeds, there is a chance that more global variables may need to be synchronised before communication is necessary. If \( n \) global variables can all be synchronised in one communication operation, \( n - 1 \) communication start-up delays are avoided. In [5] it is described how the execution time of a parallel program has been improved significantly by collecting up communication operations and only performing them when it is strictly necessary.

Figure 2.4: The time required for a reduction operation increases with the amount of data transferred
2.3 Shared variables on message-passing architectures

Consider the diagram shown in figure 2.5. In part (a) of the figure, a new value for $x$ is calculated and then $x$ is synchronised. After this, a new value for $y$ is calculated, and $y$ is synchronised. Finally, after the synchronisation of $y$, calculations involving the new value of $x$ take place. In part (b), the communication of $x$ has been delayed so that it can be synchronised along with $y$, and only one communication step occurs, reducing the communication overhead.

The reason for the performance improvements seen in [5] may be related to the fact that the amount of data transferred in a single operation is not very large (max. 500 doubles). This means that $t_s$ still is a significant term in equation (2.4). By collecting up several variables and synchronising them in one communication statement, $t_s$ becomes less significant when calculating the value for $t_c$ as $m$ increases proportionally to the number of variables being synchronised in the operation.

2.3.3 Related work

TreadMarks [7] is a software distributed shared memory (DSM) system for Unix machines. It implements a method for distributed computers to share memory pages across processors. Access to memory is granted through a locking mechanism, and release consistency [10], where updates by one processor is not visible on other processors until it releases its lock, is used to ensure a consistent view of memory for all processors. In order to save communication overhead, the library uses a lazy implementation [9] of release consistency.

It has been shown [8] that TreadMarks is capable of performing as well as hardware implementations in applications where the synchronisation frequency is moderate.

The C Region Library (CRL) [11] is a library built for the C language. In
this library data is shared between processors through user-defined regions. In order to use a shared region locally, it must be explicitly mapped into the local address space, and use of a region must be delimited by special calls to the library. Shared regions may be cached locally, reducing the communication latency of the library. CRL has been shown to perform comparably to hardware DSMs.

The main difference between the systems mentioned above and the one implemented in this project is the granularity of the shared memory. While TreadMarks implements a global address space to share memory pages and CRL has a global address space for regions of memory, the library described in this report is concerned with sharing `individual variables`. Each processor holds an individual copy of the shared variables, and the library will ensure that every shared variable has the same value on all processors. In addition to removing explicit communication statements, the library hardly requires any extra code for access to the shared variables, and the syntax for manipulating them is identical to C++ syntax. Another main difference is that the library produced by this project is able to perform run-time optimisation of the synchronisation of shared variables. Such optimisation has shown large performance improvements in the past [5], but optimising the code manually is very tedious and time consuming. Using run-time optimisation, the library has achieved the same performance as manually optimised code.

Run-time optimisation of parallel code is also used in [13], which proposes a delayed-evaluation, self-optimising (DESO) numerical library for a distributed memory parallel computer. The library is used for vector-matrix routines. By delaying the evaluation of operations, the library is able to capture the control-flow of the program, and can construct a directed acyclic graph (DAG) representing the computation to be performed. By analysing the DAG, the library is able to calculate an optimised execution plan by propagating data placement constraints backwards through the DAG. This means that the library is able to calculate a very efficient initial distribution for the data across the processors, and hence fewer redistributions of the data will be necessary.

[12] describes further issues related to the library described in [13]. The main issue described in this paper is how the library can recognise previous execution paths and reuse its previous optimisation calculations rather than having to do them again. This can be particularly useful in loops, where essentially identical DAGs are likely to occur [12].
Chapter 3

An ADT for run-time fusion optimisation

3.1 Assumptions

A number of assumptions were made about the parallel computer the ADT should work with and how it should be used in programs. The assumptions were made in order to keep the design simple, and a number of suggestions on how the assumptions can be relaxed are given in section 6.2.3. The assumptions made are as follows:

- **The ADT will work with the MPI library.** This assumption does not significantly restrict the use of the library in any way, as most message-passing architectures these days have an implementation of MPI. It is mainly included for clarity.

- **The ADT should work with the SPMD model of computing.** SPMD is described in section 2.1. This assumption is necessary as it is vital that the same global variables are declared on each processor, and when the processors run the same program, they will declare the same variables. If different global variables are declared on different processors, the behaviour of the ADT will be undefined.

- **Global variables must be initialised to the same value on each processor.** This assumption is necessary for consistency. A global variable cannot be initialised to different values on different processors, as it would break the requirement that a global variable should have the same value on all processors. If a global variable is initialised to different values, the behaviour of the ADT will be undefined.

- **If any processor performs an operation that changes the value of a global variable, then all other processors will also perform a similar operation the global variable.** This assumption
is made to keep the initial implementation simple. The reasoning is that the local value of a global variable will become “invalid”, i.e. it will need to be synchronised, when it is changed locally on a processor. If only one processor changed the value of a global variable, then extra communication would be required for the processors to agree upon which global variables are invalid before any synchronisation can take place. Also, due to the SPMD assumption, it is not difficult to ensure that this assumption is true.

• If a processor assigns a new value to a global variable, then all other processors must assign the same value to the global variable. Again, this is required for consistency. It is easy to imagine a situation where each processor assigns a completely different value to what is meant to be a global variable and thereby violating the consistency property of global variables.

3.2 Semantics of shared variables

This section contains an informal discussion of semantic rules for some operations on shared variables. The context of the discussion is shared variables of the double precision floating point type (this type will be referred to as a double for the remainder of this report). The section is intended as a feasibility study of the semantics of some operations on numbers, and the discussion can easily be extended to other data types, for example integers. There are several reasons why the double type has been chosen as a first global type to investigate. The main reasons are that most programming languages already have support for doubles, and arithmetic operations on doubles are well-understood. Also, in the case study described in section 5.4, all the global variables that are used are of this type.

As mentioned, the semantics of arithmetic operations on doubles are very well-understood for sequential programs. Difficulties arise, however, when attempting to extend the rules to apply to parallel programs, as an operation on a global double will not only be executed on a single processor, but on $p$ processors ($p \geq 1$). This project has considered some of the basic arithmetic operations and the semantic rules chosen for these operations are discussed below.

The main issues that are discussed are the difference between operations where the result is stored in a shared variable, and operations where the result is stored in a local variable. The main problem is that if the result is shared, then it is vital that each processor evaluates the same result. This leads to problem, for example when processors make local changes to a shared variable. Operations that return a local result are less of a problem as the processors can be allowed to return different values for the result.
3.2 Semantics of shared variables

a, a' - shared variables
x_i - local variables

![Diagram showing computation and communication steps]

Figure 3.1: Result of operation is stored in a global double

3.2.1 Operations where the result is a shared variable

The first rule investigated is that of adding a number to a global variable. According to the assumptions given in section 3.1, when a processor adds a number to a global variable, then all processors i (i ∈ {1, ..., p}) add a number (not necessarily the same number) to the same global variable. If each processor adds a local number x_i to a global variable a, and the result is a new global variable, a', the operation executed on each processor is a + x_i. The main problem is to decide what the global result of the operation should be. Clearly a processor cannot be allowed to see the value of the result as a + x_i as this could result in p different values for the new global variable. The solution that was chosen is that each processor will see the value of a' as:

\[ a' = a + \sum_{i=1}^{p} x_i \]

This rule means that after a processor has added a local value to a global variable, it needs to synchronise with all the other processors in order to determine the global value of the result. This process is illustrated in figure 3.1.

The next case is that of adding two global numbers together. As stated before, the operation will be executed concurrently on all p processors participating in the algorithm. There are two cases to consider, the case where two global doubles are added together and stored in a new global double, and the case where a global double is incremented by the value of another global double.

In the first case, the operation executed on each processor is a = b + c, where a, b and c are all global variables. In this case, each processor sees
the new value of $a$ as

$$a = b + c$$

As the value of both $b$ and $c$ can be agreed upon by all processors, no interprocess communication should be necessary to determine the new value of $a$.

In the second case matters are slightly different. This case deals with incrementing the value of a global variable by the value of another global variable. In C/C++ syntax, the operation executed on each processor is $a + = b$, where both $a$ and $b$ are global variables. In sequential programs, this operation is equivalent to the operation $a = a + b$, suggesting that the semantics from the case above should be used and the new value of $a$ should be $a + b$. However, the design choice was made that an increment operation should increment the global value of $a$ by the sum of all the local increments, irrespective of whether they are global variables or not. Hence, the result of the operation $a + = b_i$, where $b_i$ can be either a local value on each processor or a global variable $b$, should be ($p$ is the number of processors):

$$a = a + \sum_{i=1}^{p} b_i$$

If $b$ is a global variable, then no interprocess communication should be necessary to determine the new value of $a$. If $b$ is a local value on each processor, then the value of $a$ needs to be synchronised to determine the global change in $a$, as illustrated in figure 3.2.

3.2.2 Operations where the result is a local variable

The next rules to consider are operations involving global variables where the result is stored in a local variable on each processor. As local variables
3.3 Delaying communication

are allowed to be different on separate processors, it is perfectly legal for the results of the operations to differ on the various processors.

The global variables that participate in operations where the result is stored in a local variable should simply be treated as if their current value was a local value. Hence, the operation \( x_i = a + y_i \), where \( a \) is a global variable, may result in a different \( x_i \) for every processor \( i \), as each processor \( i \) will see the new value of \( x_i \) as \( a + y_i \). This is illustrated in figure 3.3.

3.3 Delaying communication

The advantages of delaying communication of global variables have already been discussed, and it is important that the ADT is able to exploit any opportunity for such optimisation, while preserving the semantics described above.

A scheme to delay communication can be outlined as follows: Assume that a global variable on a particular processor needs synchronisation to calculate its global value. Because of the assumptions listed in section 3.1, every processor will know that this global double needs synchronisation. Moreover, as operations are executed in the same order on all the processors (the SPMD model), global variables will acquire the need for synchronisation in the same order on every processor. These facts mean that, in order to delay communication, the ADT will only need to keep a list of all the variables that need to be synchronised. When communication is forced, the ADT should pack all the variables on the list into a single message and perform communication with all the processors participating in the algorithm.

Clearly, this scheme is dependent on the assumptions made about the environment. If the assumptions were relaxed, it might not be as straightforward to determine which variables need communication. This is discussed in section 6.2.3.
3.4 Forcing communication

At some point it will be necessary to perform all outstanding communication. These points will be referred to as force points, as they force communication to take place. Examples of force points are conditional statements and assignment of the value of a global variable to a local variable.

Synchronisation of a global variable can be delayed for as long as the global value of the variable is not required to be known. In operations only involving addition and subtraction and where the result is stored in a global variable, there is no need to force communication of any unsynchronised global variables that take part in the operation. The reason for this is that the local change in a global variable can be taken into account when determining the overall change in the resultant global variable, and hence no communication will be forced at this point. This is illustrated in figure 3.4.

A force point occurs when the global value of a variable is required, and inter-process communication is needed before it can be known. This can, for example, happen when the value of the global variable is assigned to a local variable. The reason why this operation cannot be delayed is that there is no way for the ADT to know when the local variable is used in computation, and therefore it must be assigned the correct value. Conditional statements, whose outcome determines the execution path through the program, will also be force points. For the purposes of this project, multiplication and division involving global variables also act as force points as, due to time constraints, no efficient way of delaying these operations was implemented.
Chapter 4

Implementation of a Communication Fusion Library

This section describes the implementation of a communication fusion library (CFL) that satisfies the description for the ADT outlined in chapter 3. The library is intended as a proof of concept and aims to demonstrate the benefits of removing explicit communication calls and performing communication fusion optimisations at run-time.

4.1 Choice of implementation language

A vital part of implementing a library is to choose a suitable implementation language. When deciding between the various programming languages that are available it is important to consider the type of applications that the library should work with, any libraries that the library may require to function properly, and also which language suits the particular problem best. In the case of the CFL library, C++ was chosen as the most appropriate implementation language for the following reasons:

- **The MPI library works with C++.** This was an essential requirement in order for the library to be portable, and was set out as one of the assumptions in section 3.1. As MPI implementations were only available for C/C++ and FORTRAN, this requirement restricted the choice of language considerably.

- **C++ is an object-oriented language.** It became clear early on that the object-oriented (OO) paradigm would suit a library of this type very well. OO supports mechanisms for grouping similar concepts together in classes that encapsulate their behaviour and provide a clear interface to the environment. By encapsulating each variable that
needs to behave like a global variable as an object, the user can be given a consistent and clear interface to work with.

- **C++ provides functionality for operator overloading.** Operator overloading allows the programmer to alter the behaviour of certain operations on objects. For example, it is possible for the user to define the meaning of basic operations such as ‘+’ and ‘-’ on objects. As the data types defined by the CFL should appear to behave as if they were standard variables, operator overloading is a particularly useful feature to have in the implementation language.

- **The program that motivated the creation of the CFL was written in C++.** The case study described in section 5.4 was the program that demonstrated a need for a library to perform the type of run-time optimisation that the CFL is designed to perform. As this case study would have a vital role in assessing the success of the CFL, it was important that it should be easy to integrate the two programs. By using the same implementation language, the task of linking the two programs is greatly simplified.

### 4.2 Issues on parallel libraries

A library is a set of data types and functions that can be used in a wide range of programs. The reason for building libraries is to provide support for commonly used functionality without the programmer having to implement this functionality in every single program that is written. Libraries should be made as general as possible in order to support a wide range of applications, and at the same time they should be computationally efficient.

The main problem when writing a library that performs run-time optimisation of a user’s application is that at the library’s compile-time, no knowledge of the user’s code exists. In [14], it is proposed to use lazy evaluation libraries to overcome this difficulty. A lazy library will store the operations that the user’s code requests, but will only perform the operations when it is strictly necessary. By delaying operations, the library is able to analyse the operations, and can optimise the data placement and communication that needs to take place on a parallel computer in order to suit the particular operation sequence.

An important issue to consider with libraries on message-passing architectures is the fact that both the library and the user’s application will be sending messages. It is quite likely that, if no protection is provided, the library may pick up messages that were intended for the user’s code and vice versa [1]. Doing so could have potential to break the execution of the program, and it is vital that guards are in place to prevent it from happening.
4.3 The shared double

MPI provides features that help guard against the receipt of wrong messages. A **communicator** \([1]\) encapsulates a particular context within a message-passing environment. Messages passed within one communicator cannot be picked up from another communicator. By declaring a special communicator for a library, it can be ensured that messages passed by the library are not picked up by applications, and that the library does not accidentally pick up messages intended for the application.

### 4.3 The shared double

The remainder of this chapter is concerned with the implementation of the communication fusion library and, in particular, the implementation of a shared double which mirrors the standard C++ double type. The implementation of the shared double can be used as a template for development of further shared types such as integers, etc. Section 6.2.2 has further suggestions for adding types to the library. C++-like syntax will be used in various examples when describing the implementation.\(^1\)

#### 4.3.1 The CFL_Double class

The shared double is encapsulated in a class called **CFL_Double**. Figure 4.1 illustrates this class with some of its main member variables and functions.

The first thing to notice about the **CFL_Double** is that nearly all the member functions are *private* functions, i.e. they cannot be called by any

---

\(^1\)Readers unfamiliar with the features of C++ may find [3] a useful reference, particularly the chapter about operator overloading.
other functions than *members* or *friends* of the class [3]. Having this property is important as the application programmer should not be allowed to (and should not need to) interfere with the internal workings of the library.

The only method that is declared *public* is the operator *double()* method. This method will be called when an object of type *CFL_Double* is converted to a *double*. Such type conversion will take place when e.g. a *CFL_Double* is assigned to a variable of type *double*, or when it is used in an operation that is not defined on the *CFL_Double* class, but is defined on the *double* type. The *method* will always return the current *global* value of the object, forcing communication to take place if necessary.

**Member variables**

The most important *member variables* of the *CFL_Double* class, as shown in figure 4.1, are *val*, *delta_val* and *pending_comm*. These are all *private* member variables, meaning they cannot be accessed from outside the class. The function of each of the member variables is as follows:

- *val* stores the global value of the *CFL_Double* from the last time it was synchronised. This means that immediately after a communication operation has taken place, *val* stores the current global value of the *CFL_Double*. During execution, an operation may change the value of the *CFL_Double*, and communication of the change is delayed, *val* will be *out of date* and will only become valid again *after* communication has taken place. *CFL_Double* is changed during execution, but communication of the change is delayed, the value held by *val* will no longer be the *current* global value.

- *delta_val* stores any changes made to the *CFL_Double* since the last time it was synchronised. It is tempting to believe that if *delta_val* is 0, then no changes have been made to the object. The problem is that it is possible that more than one change has occurred and they happen to have cancelled each other out on this particular processor. On other processors this may not be the case, and the object will have to be synchronised before its current global value is known.

- *pending_comm* is *true* if the global value of the object has changed since the last synchronisation. This means that the object is on a list awaiting communication before its current global value can be known. Communication is handled by the *communication manager* described in section 4.4. When the value of *pending_comm* is *false* no changes to the global value of the object has occurred since the last synchronisation, and the number stored in *val* is the current global value of the object.
4.3 The shared double

CFL_Double x;
int my_rank;

my_rank = MPI_Get_rank(&rank);
x += my_rank;
cout << x;

Figure 4.2: The effect of operations on member variables

The variables described above will change when operations are performed on the object. Consider the program in figure 4.2, written in pseudo-C++ notation. The program declares a variable x of type CFL_Double. Initially, both val and delta_val of x are 0, and pending_comm is false. The only operation performed by the program is for each processor to add its rank\(^2\) to \(x\), and then each processor prints out the value of \(x\). After the addition operation has been performed, the member variables of \(x\) have changed. Notice that val remains the same as before, but delta_val has been increased by my_rank on each processor, and pending_comm is true. When the program reaches the line where the value of x is printed to the screen, the program will attempt to convert x to a double. This will trigger a synchronisation operation to take place, as the current global value of x is not known. After the synchronisation has taken place, val has been incremented by the sum of the ranks of all the processors, delta_val has been set to 0 and pending_comm is false.

Member functions

The member function operator double() that converts a CFL_Double to a double has already been described above. This section briefly describes some of the other member functions of the CFL_Double.

- request_comm() is called when the value of the object has changed

\(^2\)The rank of a process is its id number. For \(p\) processors each processor's rank is a unique number between 0 and \(p - 1\).
and the object requires communication before its global value is known. If it is the first time the function is called after a communication step, then the object tells the communication manager (see section 4.4) that it needs synchronisation. If it is not the first time the value of the object has changed, the object is already queued for communication and no further actions are required.

- `update(double x)` is called by the communication manager after the global change in the object's value is known (i.e. after a communication step). The argument `x` is the global change in the object’s value, and is added to the member variable `val`.

- `is_valid()` returns a boolean value which is `true` if the object’s `val` variable contains the current global value for the object and `false` if not. For the purpose of the current implementation the value returned only depends on the `pending_comm` member variable, the function returning the inverse of this value. However, future extensions to the library, allowing for lazy evaluation of operations, can mean that other conditions come into play in determining whether a value is valid or not, so the function is included for forwards compatibility.

- `get()` returns the global value of the object. The function will check if the object is valid and will force communication if required before returning the global value of the object.

### 4.3.2 Implementation of operations

An important design objective for the `global_double` is to be able to perform operations on it as if it was no different than other doubles. This is achieved through the process of operator overloading [3], and this section gives a brief explanation of how some of the operator functions are implemented.

Some of the operator functions are members of the `CFL_Doube` class, while others are not. Those operator functions that are not members of the class are referred to as *helper functions* [3]. The operator functions that are members of the `CFL_Doube` class are those where the result of the operation is stored in one of the `CFL_Doubles` that participate in the operation. These operators are shown in table 4.1. The operator functions that are not members of the `CFL_Doube` class are functions in which the result of the operation is not stored in one of the `CFL_Doubles` participating in the operations. These operations are shown in table 4.2. Operator functions that are not members of the `CFL_Doube` class do not have access to the private members of the class. However, it is necessary for these functions to be able to update the member variables, so in order for them to have access to the private functions and variables of the class, the operators are declared as *friends* of the class. Overriding the traditional encapsulation principles of
4.3 The shared double

<table>
<thead>
<tr>
<th>Function</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+=</td>
<td>a += x</td>
</tr>
<tr>
<td>-=</td>
<td>a -= x</td>
</tr>
<tr>
<td>*=</td>
<td>a *= x</td>
</tr>
<tr>
<td>/=</td>
<td>a /= x</td>
</tr>
<tr>
<td>=</td>
<td>a = x</td>
</tr>
</tbody>
</table>

Table 4.1: Operations that are members of the CFL_Double class

<table>
<thead>
<tr>
<th>Function</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>x = a + y</td>
</tr>
<tr>
<td>-</td>
<td>x = a - y</td>
</tr>
<tr>
<td>*</td>
<td>x = a * b</td>
</tr>
<tr>
<td>/</td>
<td>b = y / a</td>
</tr>
<tr>
<td>+=</td>
<td>x += a</td>
</tr>
<tr>
<td>-=</td>
<td>x -= y + b</td>
</tr>
<tr>
<td>*=</td>
<td>x *= a - b</td>
</tr>
<tr>
<td>/=</td>
<td>x /= b</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>cout &lt;&lt; a</td>
</tr>
</tbody>
</table>

Table 4.2: Operations that are not members of the CFL_Double class

the OO paradigm, this gives the functions complete access to all the private members of the class.

Passing CFL_Doubles as argument to operator functions

When passing a CFL_Double as an argument to an operator function or returning a CFL_Double from any function, it is necessary to use the pass by reference feature of C++. This is done by adding an ‘&’ at the end of the type name in the function declaration. For example, the function

    CFL_Double& operator+=(CFL_Double& a, double b)

takes a reference to a CFL_Double and a copy of a double as arguments and returns a reference to a CFL_Double. This means that the function will have direct access to the object and not a copy of it. Pass by reference is discussed further in [3].

Local CFL_Doubles

Occasionally an operation will require intermediate results to be stored for future use. For example, the operation \( a = b + c \) will require the result of adding \( b \) to \( c \) to be calculated and stored before it is assigned to \( a \). The
CFL_Double b = 29;
double c = 12;

a = b + c;

Evaluate \( b + c \) first according to precedence

Return 41
Return \( 29 + (\text{no}_\text{procs}) \times 12 \)

Figure 4.3: The outcome of an operation may not be known until the type of the result is known

temporary value will be stored in a CFL_Double object, and CFL_Doubles that store temporary results will be referred to as local CFL_Doubles. The use of local CFL_Doubles means that evaluation of operations is not always straightforward.

The problem arises if, for example, \( b \) and \( c \) are of type CFL_Double and \( a \) is a normal double. If \( b \) or \( c \) require communication before their global value is known, then communication will have to take place before the value of the result can be assigned to \( a \). However, as it is desirable to delay communication for as long as possible, \( b \) and \( c \) will not be automatically synchronised by the '+' operator. Any synchronisation required will be forced by the '=' operator, which is evaluated after the '+' operator.\(^3\)

Another problem with the above operation arises if \( b \) is a CFL_Double and \( c \) is a double. In this case the result of the '+' operation will be different depending on whether \( a \) is a CFL_Double or a double. This problem is illustrated by figure 4.3, where, even when the global value of \( b \) is known, the '+' operator cannot know what the result of the operation should be.

In order to solve the problem, another member variable, local_delta, was introduced into the CFL_Double class. As the '+' operator is only able to see \( b \) and \( c \) it has to make an intermediate decision on how to proceed based on the types of these variables. If both are CFL_Doubles, the outcome of the operation is known as it does not depend on the type of the result. However, if \( c \) is a double, '+' will return a new CFL_Double that is identical

\(^3\)C++ operator precedence rules give higher precedence to '+' than to '=', and it is not possible to redefine these rules [3].
4.4 The communication manager

The communication manager takes care of all the bookkeeping for the CFL, deals with all interprocess communication and ensures that the correct CFL_Doubles are updated after communication has taken place. The communication manager is encapsulated by a class called CFL_Manager, and the most important member variables and functions of this class are shown in figure 4.5.

The variables and functions in the CFL_Manager class have the following purposes:

- **comm_list** is the list of CFL_Doubles that need to be synchronised.

![Diagram](image)

Figure 4.5: The CFL_Manager class
before their global value is known.

- init(MPI_Comm c) initialises the CFL_Manager. It takes an MPI communicator as an argument, and the CFL_Manager will manage all global variables within this communicator. In order to avoid the problems discussed in section 4.2, the CFL_Manager will duplicate the communicator to get a different, but identical, communicator to send its messages in.

- add_comm_list(CFL_Double *a) adds the CFL_Double pointed to by a to the comm_list. The next time communication is forced, this CFL_Double will be synchronised.

- rm_comm_list(CFL_List *l) removes the CFL_Double in position l from the comm_list. This function is called by a CFL_Double if it no longer needs synchronisation, for example if its value is overwritten by an assignment operation.

- force() forces communication to take place and updates the value of all the CFL_Doubles on the comm_list with their current global value.

4.4.1 Delaying communication

When an operation decides that a CFL_Double needs communication before its global value is decided, it will call the request_comm member function of the CFL_Double class. This function will check that the object has not requested communication before, and will then call the add_comm_list member function of the CFL_Manager. The CFL_Manager will concatenate the CFL_Double to the front of its comm_list, and will also tell the CFL_Double where it is stored in the list (i.e. what memory position it is stored at) so that it is easy to remove it from the list if that should prove necessary.

4.4.2 Forcing communication

When a force point is reached in an operation, the force() function of the communication manager is called. The following events are illustrated by figure 4.6. As shown in the figure, the communication manager will inspect its comm_list and store the change in each CFL_Double in the list (i.e. the delta_val) in an array. Having done so, the MPI_Allreduce [1] function will be called, and the change for each CFL_Double will be summed across all processors and stored in a new array on each processor. As the MPI_Allreduce operation requires all processors to be synchronised, some processors may have to wait for others to catch up before the operation can take place. Once the reduction operation is complete, each CFL_Double on the comm_list will be updated with the global change in its value, and execution of the program will be allowed to proceed.
4.4 The communication manager

Application | Communication Manager

- Force comm
- Marshall all delta values into an array
- Perform the communication operation
- Update CFL_Doubles with global delta values
- Continue program

Figure 4.6: The events subsequent to forcing communication
Chapter 5

Evaluation

Extensive testing of the CFL has been carried out to ensure that it works as specified. All testing and performance evaluation was done on the Fujitsu AP3000 parallel computer owned by the Imperial College Parallel Computing Centre. This chapter describes the testing process and gives the results from the performance evaluation of the CFL.

5.1 The Fujitsu AP3000 parallel computer

The Fujitsu AP3000 is a massively parallel distributed memory computer consisting of 80 processors. 60 of these processors are 300MHz 64-bit UltraSPARC processors, and there are also 16 200MHz UltraSPARC and 4 200MHz UltraSPARC dual processors. The computer has a total of 12.5GB of main memory and 764GB of disk storage [15].

The processors on the AP3000 are connected via a dedicated high-speed network (AP-Net). The network topology is a 2D mesh, and the links are bidirectional with a bandwidth of 200MB/s in each direction. Figure 5.1 illustrates the interconnection network.

Jobs on the AP3000 can either be run interactively or as batch jobs. Jobs run on interactive nodes will not have exclusive access to the processors they run on. In order to gain exclusive access to processors (e.g. for performance evaluation), a job must be submitted as a batch job to a queueing system. The queueing system will only schedule a job when the requested number of processors is available, and the job will have exclusive access to the processors it is run on throughout its execution.

A wide range of programming tools are supported by the AP3000. Importantly for this project, both a version of the GNU C++ compiler and an implementation of the MPI libraries are provided.
5.2 A test harness for the CFL

A large suite of test cases was built in order to ensure the correctness of the CFL. The test suite used for this purpose consists of approximately 8500 lines of code, covering 376 test cases.\(^1\)

The test suite consists of a main loop that calls the various test cases in turn. Each test case consists of a case function and an answer function. The case function uses the CFL library to perform some arithmetic operation involving global variables. The result of the operation is returned by the function and is compared to the result of the answer function, which calculates the result without using the CFL library. If these results are not identical, the test program will print out which test case that failed, and a summary of the total number of cases covered and number of failures is given at the end of testing.

The test suite will run on any number of processors, and the initial values of the numbers used in the test cases can be reconfigured easily by the user. The execution time of the test suite is negligible, and it has proved invaluable in debugging the various execution paths through the CFL library. The test suite is not complete, due to time constraints making it infeasible to implement an exhaustive suite of test cases. The development of a full test suite is left as future work.

\(^1\)Appendix A gives a complete listing of all the test cases
5.3 Evaluation of small examples

double x, a, b, n;

for(i=0; i<NO_ITER; i++)
{
    a = i*(rank+1)*0.0001;
    MPI_Allreduce(&a, &b, 1, MPI_DOUBLE, MPI_SUM,
                   MPI_COMM_WORLD);
    x += b;
}

n += x;

Figure 5.2: Naive code for the first example

5.3 Evaluation of small examples

In order to perform a preliminary evaluation of the performance impact and the usability of the CFL, a number of small test programs were implemented. Each test program was implemented in three versions: a naive, unoptimised version, a manually optimised version and a version where the optimisation was performed automatically by the CFL at run-time.

Each test case consists of a for-loop that is executed a given number of times. As the programs are run on varying numbers of processors, it was ensured that the sum of the iterations through the for-loop was the same for all the runs.

Example 1

In the first example, $x$ is a shared variable. In each iteration of the for-loop, $x$ is incremented by a certain value, and after the for-loop has completed, the local variable $n$ is incremented by the value of $x$. In the naive example, one reduction operation takes place for every iteration of the for-loop. The C++ code for the loop is given in figure 5.2.

Manual optimisation of this example is relatively easy. Instead of updating the value of $x$ at every iteration through the loop, it is possible to move the reduction call outside the loop and simply calculate the entire local change in $x$ before summing it across all the processors. The code for the optimised loop is shown in figure 5.3.

By declaring $x$ as a CFLDouble, the code for the example is simplified even further, and also all explicit communication is removed from the code. It is much easier to see that it is the value of $x$ that is being incremented in the loop as the need for temporary variables to store values in is eliminated. The code for the loop using the CFL library is given in figure 5.4.

Timing results against number of processors for this example are shown in figure 5.5. The graph shows the advantages of optimising the commu-
double x, a, b, n;

for(i=0;i<NO_ITER;i++)
{
    a *= i*(rank+1)*0.0001;
}

MPI_Allreduce(&a, &b, 1, MPI_DOUBLE, MPI_SUM,
              MPI_COMM_WORLD);

x *= b;
n *= x;

Figure 5.3: Manually optimised code for the first example

CFL_Doule x;

double n;

for(i=0;i<NO_ITER;i++)
{
    x *= i*(rank+1)*0.0001;
}

n *= x;

Figure 5.4: Code for the first example using the CFL library

Figure 5.5: Timing results for example 1
**5.3 Evaluation of small examples**

![Timing results for optimised runs in example 1](image)

**Figure 5.6: Timing results for optimised runs in example 1**

Communication pattern by moving the communication operation outside the loop. As it is impossible to tell the two optimised versions apart in this graph, figure 5.6 shows results for only the optimised runs. Allowing the CFL to perform the optimisation automatically yields a very similar timing graph to the automatically optimised case, while the execution time for the naive case illustrates the performance effect of communication by increasing the execution time when increasing the number of processors. The timing results for one processor demonstrates that, as expected, there is an overhead associated with using the CFL, but as the number of processors increases, this overhead becomes less significant. A possible explanation is that, as the for-loop is executed the same number of times for all executions, the total overhead remains constant but is spread across the processors participating in the algorithm.

**Example 2**

In the second example, the for-loop calculates changes for three shared variables, \(x, y,\) and \(z,\) and the local variable \(n\) is incremented by the value of these three variables for each iteration of the loop. The amount of computation performed is quite small compared to the amount of communication, meaning that a relatively large proportion of the execution time will be communication time.

In the naive case, a reduction operation takes place after the change in each shared variable has been calculated, resulting in three reduction operations per iteration of the for-loop. The code for the naive example is
double x, y, z, a, b, c, d, e, f, n;

for(i=0;i<NO_ITER;i++)
{
    a = i*(rank+1)*0.0001;
    MPI_Allreduce(&a, &d, 1, MPI_DOUBLE, MPI_SUM,
                  MPI_COMM_WORLD);
    x += d;

    b = (i+3)*(rank+4)*.0002;
    MPI_Allreduce(&b, &e, 1, MPI_DOUBLE, MPI_SUM,
                  MPI_COMM_WORLD);
    y -= e;

    c = (-i-1)*(rank+3)*0.000023;
    MPI_Allreduce(&c, &f, 1, MPI_DOUBLE, MPI_SUM,
                  MPI_COMM_WORLD);
    z += f;

    n += x + y + z;
}

Figure 5.7: Naïve code for the second example

given in figure 5.7.

Using manual optimisation, it is possible to enhance the code so that only one reduction operation takes place in each iteration of the loop. This is done by calculating the change in all three shared variables before doing any communication, and doing reduction on all three variables in one operation. The code for the manually optimised example is shown in figure 5.8.

By declaring x, y and z as CFL_Doubles, the code for the example is simplified even further. No explicit communication or temporary variables is required anymore, and optimisation will be performed by the CFL at run-time. The code using the CFL is shown in figure 5.9.

A graph of the execution time against number of processors for this example is shown in figure 5.10. Again, the results show the impact of increased communication on execution time, increasing the execution time dramatically when communication is introduced (i.e. more than one processor participates in the algorithm). As there is very little computation being carried out in this example, increasing the number of processors actually slows down the code rather than speeding it up. However, the naïve implementation shows a more dramatic slowdown than the two optimised implementations, due to the fact that less communication takes place in the
5.3 Evaluation of small examples

double x, y, z, a[3], b[3], n;

for(i=0; i<NO_ITER; i++)
{
    a[0] = i*(rank+1)*0.0001;
    a[1] = (i+3)*(rank+4)*.0002;
    a[2] = (-i-1)*(rank+3)*0.000023;
    MPI_Allreduce(a, b, 3, MPI_DOUBLE, MPI_SUM,
                  MPI_COMM_WORLD);
    x += b[0];
    y -= b[1];
    z += b[2];

    n += x + y + z;
}

Figure 5.8: Manually optimised code for the second example

CFL_Double x, y, z;
double n;

for(i=0; i<NO_ITER; i++)
{
    x += i*(rank+1)*0.0001;
    y -= (i+3)*(rank+4)*.0002;
    z += (-i-1)*(rank+3)*0.000023;
    n += x + y + z;
}

Figure 5.9: Code for the second example using the CFL
optimised versions. It is also worth noting that the code using the CFL appears a lot more readable than the manually optimised code. It is an easy task to analyse the change each processor makes to the global variables in figure 5.9, bearing in mind the semantics for these operations. In figure 5.8, more analysis is required to discover that $b$ contains the result of the reduction operation on $a$ which initially contained the local change for each global variable.

**Example 3**

The third example investigated is very similar to the second, but is different in that local computation is introduced into the for-loop. Between the update of each variable, a small loop is iterated through to simulate the effect of local computation in order to make the example more realistic. As before, the naive example has three reduction operations in each iteration of the loop, and the code is shown in figure 5.11.

The code can be manually optimised in a similar way to the second example, resulting in only one communication operation for each iteration through the loop. The manually optimised code is shown in figure 5.12.

By declaring $x$, $y$ and $z$ as CFL Doubles and removing all explicit communication calls, the CFL responsible for performing all the necessary optimisations. The code with the CFL is shown in figure 5.13.

Timing results for example 3 against number of processors are shown in figure 5.14. This shows a more realistic timing graph than the two previous examples, where the execution time is reduced as the number of processors
double x, y, z, a, b, c, d, e, f, n;

for(i=0; i<NO_ITER; i++)
{
    a = i*(rank+1)*0.0001;
    MPI_Allreduce(&a, &d, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    x += d;
    local_computation_loop();
    b = (i+3)*(rank+4)*.0002;
    MPI_Allreduce(&b, &e, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    y += e;
    local_computation_loop();
    c = (-i-1)*(rank+3)*0.000023;
    MPI_Allreduce(&c, &f, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    z += f;
    local_computation_loop();

    n += x + y + z;
}

Figure 5.11: Naive code for the third example
double x, y, z, a[3], b[3], n;

for(i=0;i<NO_ITER;i++)
{
    a[0] = i*(rank+1)*0.0001;
    local_computation_loop();
    a[1] = (i+3)*(rank+4)*.0002;
    local_computation_loop();
    a[2] = (-i-1)*(rank+3)*0.000023;
    local_computation_loop();

    MPI_Allreduce(a, b, 3, MPI_DOUBLE, MPI_SUM,
                  MPI_COMM_WORLD);
    x += b[0];
    y += b[1];
    z += b[2];

    n += x + y + z;
}

Figure 5.12: Manually optimised code for the third example

CFL_Double x, y, z;
double n;

for(i=0;i<NO_ITER;i++)
{
    x += i*(rank+1)*0.0001;
    local_computation_loop();
    y += (i+3)*(rank+4)*.0002;
    local_computation_loop();
    z += (-i-1)*(rank+3)*0.000023;
    local_computation_loop();

    n += x + y + z;
}

Figure 5.13: Manually optimised code for the third example
increases. Using equation 2.1, the speedup speedup for the runs can be calculated and is shown in figure 5.15. The speedup graph clearly demonstrates the advantage of optimising the communication pattern in the code. It is also worth noting that it becomes very difficult to tell apart the two optimised versions, which seems to indicate that the CFL performs its job very well.

The performance of the CFL in these three examples has been encouraging. It has achieved its task of making the code more readable by removing explicit communication statements, and the results produced by the code using the CFL were identical to the results produced by the original code. Also, the CFL has been able to optimise the communication patterns, delaying communication where possible.

5.4 Case study: Virtual Plankton Ecology

The examples discussed in the previous section have been constructed for the particular purpose of being able to integrate with the CFL, and as such do not say very much about how well the CFL will integrate with a real-life application. This case study investigates the performance impact of integrating the CFL with a large-scale application built for simulating plankton ecology.
5.4.1 Introduction

The bulk of the biomass in the top kilometre of the sea is made up of plankton. The concentration of plankton in the sea varies with both time and place, but concentrations as high as hundreds of millions of plankton per litre of water have been measured. Plankton affect many important aspects in the world, such as climate and pollution [16], and is also a primary source of food for fish.

Plankton can be divided into two main groups, phytoplankton and zooplankton. Phytoplankton are microscopic plants. They do not have the ability to swim, and simply drift wherever water turbulence carries them. Phytoplankton get their energy through photosynthesis, where they absorb carbon dioxide and light, and release oxygen. The speed of this process is determined by the amount of light available and the amount of nutrients in the water. Zooplankton are animals, and unlike phytoplankton they have the ability to swim vertically. Zooplankton get their energy by feeding on phytoplankton, and they are predated by fish larvae.

As plankton play a significant role in the worldwide ecosystem, it is important to have an understanding of how the plankton ecosystem works, in order to be able to predict the concentrations of various species of plankton with varying conditions. Oceanographers have worked on this problem over many years, but scientific progress has been slow due to the extremely high cost of sampling data [16].

With the emergence of fast computers, a new way has been found to generate large amounts of data. Instead of analysing samples collected at
5.4 Case study: Virtual Plankton Ecology

Figure 5.16: The water column divided into biological and physical layers

sea, scientists can simply set the parameters they want and then let the computer calculate all the required data in a short amount of time. However, due to the vast complexity of plankton ecology, it is not a simple problem for a computer to solve.

5.4.2 The ZB Model

The ZB model simulates the plankton ecology in a single $1 \times 1 \times 300$ m$^3$ column of water as it drifts through the sea, following a path pre-calculated using a utility for simulating water currents. The water column is discretised into two sets of layers, biological layers and physical layers. An illustration of the division is shown in figure 5.16.

Each biological layer contains all the biological properties of the water column at the depth of that layer. The most important feature contained in the biological layers are the various plankton and the nutrients they feed upon. In the model, plankton are grouped together in plankton bundles or particles, where each particle behaves as an individual, but actually represents thousands of individuals for efficiency reasons.

The physical layers contain the physical properties of the water column at the given depth. As the figure shows, the physical layers near the surface are quite small, while the physical layers near the bottom are large. This is due to the fact that near the surface the physical properties of the water changes very rapidly with depth, while this is not the case on greater depths.
Parallelisation of the ZB model

One of the most important issues that needs to be addressed when implementing a parallel version of the ZB model is what scheme should be used for distributing the various plankton particles across the processors. In [17] the use of horizontal partitioning of the biological layers between the processors is investigated. In this scheme, each processor is allocated a set of biological layers and performs the computations necessary for the particles that reside in those layers. The layers can be either block-allocated or stripe-allocated as illustrated in figure 5.17.

The advantage of using block allocation is that as particles migrate between the layers that reside on the same processor, no inter-processor communication is required. Only when particles move from a layer on one processor to a layer on another processor is communication required. However, as the plankton tend to reside mostly in the upper layers, block allocation tends to result in poor load balancing between the processors [5]. The load-balancing problem is alleviated somewhat by using stripe allocation, but this scheme means that inter-process communication is necessary every time a plankton particle moves from one layer to another, resulting in large communication overhead. The main problem with all horizontal partitioning schemes, however, is that the amount of computation required increases proportionally to the number of plankton particles used in the simulation.

When using vertical partitioning, each processor is statically allocated a set of particles and performs the computations required for these particles in each time step of the simulation. Vertical partitioning of the water column is illustrated in figure 5.18. Each processor now holds local data about the entire water column, and at every time step this data needs to be
synchronised across all processors. However, during particle movement and energy computations, no communication between processors is necessary. This means that instead of increasing with the number of plankton particles, the communication overhead will now increase slowly with the number of processors the simulation is run on [5]. Due to superior performance, vertical partitioning was chosen as the allocation strategy in [5], and it is the code from this project that has been used as a base for this case study.

Manual optimisation of the parallel code

In order to make the ZB model run efficiently on a parallel computer, several alterations to the communication structure proved necessary. These changes are described in detail in [5], but a brief outline is given below.

The simulation code is a for-loop that is iterated through a given number of times during a simulation run. Each iteration of the loop represents a single time step in the simulation. Throughout the loop, the properties of the water column change, due to plankton moving from one biological layer to another, nutrient uptake and other factors. For example, the amount of nitrogen in a layer is an important property of the water column, and this will change after phytoplankton has been updated.

It is important that all processors see the same values for the parameters of the water column. In order to ensure this, the values of the parameters were synchronised every time they were changed. In [5], the code is optimised by manually delaying communication and collecting up variables that need synchronisation in order to save communication operations. This process is similar to that illustrated in figure 2.5. By changing the type of the parameters that are shared between processors to CFL_Doubles, the CFL
nitrogen_required = calculate_nitRequirement();
total_nit_req = sum_across_processors(nitrogen_required);
Nitrogen -= total_nit_req;

Figure 5.19: Old mechanism for maintaining global value for nitrogen

should be able to perform similar optimisation automatically without the need for explicit communication calls.

5.4.3 Incorporating the CFL into the ZB Model

On large problems, the majority of the execution time of the ZB model is spent doing biological calculations [5]. The ZB model is a substantial piece of code, and due to time constraints it would be very difficult to incorporate the CFL into the whole model, as it takes time to read and understand the code and determine which variables can be abstracted by the CFL. For these reasons it was decided to evaluate the performance impact of incorporating the CFL into the biological part of the model first, in order to ensure that some results for real-life problems were produced.

The biological parameters of the water column are stored in the biological layers. It was therefore natural to examine these layers in order to find any variables that were shared across the processors. A large number of variables are declared for each biological layer, but it turns out that only a very small proportion of these are actually used as shared variables.

The amount of nitrogen and ammonia in each biological layer are examples of two such shared variables. These parameters are used when computing energy uptake for plankton, and it is important that every processor sees the same amount of nitrogen and ammonia in each biological layer. Before the CFL was incorporated, the variables were kept synchronised by first calculating the change in each variable on every processor, then performing a global sum on the change and then each processor would add the global change to its value for the variable, as illustrated for nitrogen in figure 5.19. This mechanism means that every time the amount of e.g. nitrogen changes, it is necessary to insert an explicit communication call into the code before the value for nitrogen can be updated, making the code harder to write and harder to understand. Manual communication fusion optimisation had been performed on this code, meaning that communication calls were collected up and delayed until they were deemed necessary by the programmer. As the global change in a value is not known until communication has taken place, the programmer also had to manually delay the update of the shared value until communication had been performed, making the code even harder to maintain.
5.4 Case study: Virtual Plankton Ecology

Many of these problems are solved by declaring nitrogen and ammonia to be `CFL_Double`. Now the programmer is able to simply add any local change to the variables without worrying about synchronisation issues. By making ammonia a `CFL_Double`, it was possible to remove 5 explicit communication calls from the code, and making nitrogen a `CFL_Double` eliminated a further 3. The function calls to update the values of ammonia and nitrogen that were previously delayed until after communication had taken place were inserted into the code where they logically belong, meaning that the CFL would have to recognise the optimisation opportunities that exist within the code by itself.

Correctness of model after changes

Having made changes to the model, it was necessary to ensure that these changes did not affect the results produced by the model. As the changes made to the code only alters the communication patterns of the code, the outcome of the computations should not be changed at all by incorporating the CFL into the code. This was verified by checking that the new code produced identical results to the original.

5.4.4 Performance of the CFL in the ZB Model

Timing results for the execution of the ZB model before and after the CFL was incorporated are shown in figure 5.20. Using equation 2.1, approximating the one processor run for each case as the best sequential execution
time, the speedup for the execution runs was calculated and is shown in figure 5.21.

The results shown in the two figures are very encouraging. In fact, the version of the code with the CFL actually outperforms the original code when it comes to execution time on a small number of processors. The speedup graphs are also quite similar, indicating that it is difficult to determine which version gives the better performance. However, it would be unfair to draw any general conclusions from these results. The reasons for this are as follows:

- The runs are made on a quite small problem size, only 640 plankton particles per biological layer. In order to evaluate the performance fully it would be necessary to investigate the effect of larger problem sizes on the execution time.

- The runs only simulate a time interval of ten days. Longer runs will be necessary to be able to draw conclusions from the timing results. Unfortunately, due to time constraints, such long runs were not performed during this project (some runs may take up to 8 hours to complete).

- Not the whole ZB model has been optimised with the CFL. As the code stands, it is a hybrid between automatic and manual optimisation. It would be desirable to incorporate the CFL into the whole ZB model before investigating the performance of the model further.
Chapter 6

Conclusion and Further Work

6.1 Conclusion

This project has designed and built a prototype library that is capable of performing run-time communication fusion optimisation on variables in parallel programs. The library works by allowing variables defined by the programmer to be shared between the processors. This makes the user’s application code easier both to write and to understand by eliminating the need for explicit communication calls to keep shared variables synchronised, and by making changes to shared variables more explicit. By delaying communication until a force-point is reached, the library is able to perform run-time optimisation by collecting together several variables that need communication and synchronising these in a single communication operation, thus saving execution time in form of communication overheads.

The preliminary performance evaluation of the library was performed on small test examples. Using run-time optimisation, the automatically optimised programs achieved similar performance to manually optimised versions of the programs. The library was also tested on a large-scale application for plankton modelling, also here matching the performance of manually optimised code.
6.2 Further work

The proof of concept implementation of the shared memory library presented in this report has shown quite encouraging results, and taking the work further will be an interesting and worthwhile process. This section discusses some aspects of the library that can be developed further.

6.2.1 Further evaluation of the CFL

It is desirable to complete the performance evaluation of the CFL on the ZB model by incorporating it into the whole model. This will enable a complete evaluation of a real-life application that relies totally on the CFL to perform all the inter-processor communication required. The CFL would have to find all opportunities for optimisation of the communication without help from the application programmer. The performance should be compared to the results obtained in [5], in order to determine the success of the CFL.

It is also important that timing results for large, realistic problems running over a long time is obtained. These results will give a more reliable picture of the real performance of the CFL than the examples given in chapter 5 of this report.

Finally, it should be determined if the code for parallel application does become easier to write and understand when using the CFL. For the examples given in this report, this seems to be the case, but a more thorough study of a range of applications would be desirable.

6.2.2 Adding data types to the CFL

This project only implemented a single shared data type, the CFL<Double>. In order for the library to become truly usable, it will be necessary to implement more of the basic data types, and also to provide support for user-defined data types. The latter will be a particularly challenging task as the user needs to specify the operations that can be performed on the data types and the semantics of these operations.

One way of supporting more basic data types could be to implement a C++ template type. A class template takes a type as a parameter in the definition of the class. This means that, for example, a template class CFL_Number could be implemented, and then particular objects of the class could be instantiated as integer or double depending on what the user wants. For an in-depth discussion of templates, see [3].

6.2.3 Relaxing the assumptions made by the CFL

The assumptions made before designing and implementing the CFL restrict its usefulness to some extent. It would be desirable to relax these assumptions so that the programmer will not have to worry about them when
writing code that uses the CFL. This section contains suggestions on how to overcome some of the assumptions made by the CFL.

- By making a single processor responsible for initialisation of shared values, the risk of a variable being initialised to different values on each processor will be eliminated. A global variable would be initialised by this processor and the initialised value would be sent to all other participating processors. This would guarantee that all processors initialise the shared variable to the same value, but would introduce some communication overhead. In a similar way, the problem with assigning new values to shared variables can be solved.

- In order to get around the assumption that all processors change a shared variable at the same time, two schemes can be used. One possible solution is a scheme where there is a message exchange before synchronising variables in order to agree upon which variables need synchronisation. This would mean that at least one extra global communication operation is necessary for each synchronisation operation. Another solution that is possible would be to simply synchronise all shared variables every time communication is forced. This scheme would also introduce a communication overhead, the size of which will be determined by the number of shared variables in a program. There is also the issue of shared variables that are declared within functions. These will only have limited scope and will need to notify the communication manager when they come into existence and when they disappear from the scope.

6.2.4 Including point-to-point messaging in the CFL

At the moment, the CFL only performs communication fusion on global communication operations where all the processors participate. It would be useful if it could perform the same type of optimisation on messages between individual processors.

During execution of parallel algorithms, it is common for messages to be exchanged between individual processors. Often, these messages are asynchronous and are only picked up at certain times by the receiving processor. However, the sending processor may send many messages between each time a receiving processor decides to inspect its message queue, as illustrated in figure 6.1.

If the SPMD model is used, the sending processor is able to predict approximately when the receiving processor is likely to inspect its message queue. The receiving processor should therefore be able to buffer the messages it wishes to send until just before the receiver is about to check for incoming messages. By sending all the data in one message, communication start-up time can be saved, as illustrated in figure 6.2.
Only one message is sent from each processor, message contains processor’s entire message buffer.

Figure 6.1: Point-to-point messages being sent between two processors

Figure 6.2: Optimised point-to-point messages being sent between two processors
6.2 Further work

Another way to implement the scheme would be for the receiving process to notify any potential senders that it is about to check for incoming messages and that any buffered data should be sent. However, this would add extra communication overhead to the system and the receiving process may have to spend some time waiting for all the incoming messages to come in.

6.2.5 Multi-threading

By running the CFL in a different thread to the application, further scope for optimisation could be found. CFL should be able to detect when data in the main application needs synchronisation, and this could be done behind the scenes without any intervention from the application.

Such a scheme could be able to hide the latency associated with forcing communication in the CFL at the moment. Currently when communication is forced, the main application will be stalled until the CFL has synchronised all shared variables. If the CFL was run in a separate thread using asynchronous communication, it will be able to perform the synchronisation of variables while the application is still doing useful work. That way a variable may already be up to date when the application wishes to use it again.
Bibliography


[16] Introductory web-pages for Virtual Plankton Ecology at the department of Earth Resources Engineering, Imperial College, URL: http://www.ere.ic.ac.uk/research/or/introduct.htm

Appendix A

Cases covered by the test harness

This appendix lists the test cases covered by the test harness. The following notation is used when describing a test case:

- $v$ - a valid global double (i.e. pending_comm is false).
- $pc$ - an invalid global double.
- $d$ - a local double

A subscript $l$ after a global double means that is is a $local$ CFL_Double, as described in section 4.3.2.

The test cases are as follows:

<table>
<thead>
<tr>
<th>Case</th>
<th>Description</th>
<th>Case</th>
<th>Description</th>
<th>Case</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v += v$</td>
<td>$v = v + v$</td>
<td>$v = v - v$</td>
<td>$v += pc$</td>
<td>$v = v + pc$</td>
<td>$v = v - pc$</td>
</tr>
<tr>
<td>$pc += v$</td>
<td>$v = pc + v$</td>
<td></td>
<td>$pc += pc$</td>
<td>$v = pc + pc$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$v -= v$</td>
<td>$pc = v + v$</td>
<td>$pc = v - v$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$v -= pc$</td>
<td>$pc = v - pc$</td>
<td></td>
</tr>
<tr>
<td>$pc -= v$</td>
<td>$pc = pc - v$</td>
<td>$pc = pc - v$</td>
<td></td>
<td>$pc -= pc$</td>
<td>$pc = pc - pc$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>49</td>
<td>$v += v + v$</td>
<td>50</td>
<td>$v += v + pc$</td>
<td>51</td>
<td>$v += pc + v$</td>
</tr>
<tr>
<td>53</td>
<td>$pc += v + v$</td>
<td>54</td>
<td>$pc += v + pc$</td>
<td>55</td>
<td>$pc += pc + v$</td>
</tr>
<tr>
<td>57</td>
<td>$v += v - v$</td>
<td>58</td>
<td>$v += v - pc$</td>
<td>59</td>
<td>$v += pc - v$</td>
</tr>
<tr>
<td>61</td>
<td>$pc += v - v$</td>
<td>62</td>
<td>$pc += v - pc$</td>
<td>63</td>
<td>$pc += pc - v$</td>
</tr>
<tr>
<td>65</td>
<td>$v -= v + v$</td>
<td>66</td>
<td>$v -= v + pc$</td>
<td>67</td>
<td>$v -= pc + v$</td>
</tr>
<tr>
<td>69</td>
<td>$pc -= v + v$</td>
<td>70</td>
<td>$pc -= v + pc$</td>
<td>71</td>
<td>$pc -= pc + v$</td>
</tr>
<tr>
<td>73</td>
<td>$v -= v - v$</td>
<td>74</td>
<td>$v -= v - pc$</td>
<td>75</td>
<td>$v -= pc - v$</td>
</tr>
<tr>
<td>77</td>
<td>$pc -= v - v$</td>
<td>78</td>
<td>$pc -= v - pc$</td>
<td>79</td>
<td>$pc -= pc - v$</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>81</td>
<td>$v += v + d$</td>
<td>93</td>
<td>$v += v - d$</td>
<td>105</td>
<td>$v -= v + d$</td>
</tr>
<tr>
<td>82</td>
<td>$v += pc + d$</td>
<td>94</td>
<td>$v += pc - d$</td>
<td>106</td>
<td>$v -= pc + d$</td>
</tr>
<tr>
<td>83</td>
<td>$v += d + v$</td>
<td>95</td>
<td>$v += d - v$</td>
<td>107</td>
<td>$v -= d + v$</td>
</tr>
<tr>
<td>84</td>
<td>$v += d + pc$</td>
<td>96</td>
<td>$v += d - pc$</td>
<td>108</td>
<td>$v -= d + pc$</td>
</tr>
<tr>
<td>85</td>
<td>$pc += v + d$</td>
<td>97</td>
<td>$pc += v - d$</td>
<td>109</td>
<td>$pc -= v + d$</td>
</tr>
<tr>
<td>86</td>
<td>$pc += pc + d$</td>
<td>98</td>
<td>$pc += pc - d$</td>
<td>110</td>
<td>$pc -= pc + d$</td>
</tr>
<tr>
<td>87</td>
<td>$pc += d + v$</td>
<td>99</td>
<td>$pc += d - v$</td>
<td>111</td>
<td>$pc -= d + v$</td>
</tr>
<tr>
<td>88</td>
<td>$pc += d + pc$</td>
<td>100</td>
<td>$pc += d - pc$</td>
<td>112</td>
<td>$pc -= d + pc$</td>
</tr>
<tr>
<td>89</td>
<td>$d += v + d$</td>
<td>101</td>
<td>$d += v - d$</td>
<td>113</td>
<td>$d -= v + d$</td>
</tr>
<tr>
<td>90</td>
<td>$d += pc + d$</td>
<td>102</td>
<td>$d += pc - d$</td>
<td>114</td>
<td>$d -= pc + d$</td>
</tr>
<tr>
<td>91</td>
<td>$d += d + v$</td>
<td>103</td>
<td>$d += d - v$</td>
<td>115</td>
<td>$d -= d + v$</td>
</tr>
<tr>
<td>92</td>
<td>$d += d + pc$</td>
<td>104</td>
<td>$d += d - pc$</td>
<td>116</td>
<td>$d -= d + pc$</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>129</td>
<td>$v = v + v_l$</td>
<td>145</td>
<td>$v = v - v_l$</td>
<td>161</td>
<td>$v = v + v_l$</td>
</tr>
<tr>
<td>130</td>
<td>$v = pc + v_l$</td>
<td>146</td>
<td>$v = pc - v_l$</td>
<td>162</td>
<td>$v = pc + v_l$</td>
</tr>
<tr>
<td>131</td>
<td>$v = v + pc_l$</td>
<td>147</td>
<td>$v = v - pc_l$</td>
<td>163</td>
<td>$v = v + pc_l$</td>
</tr>
<tr>
<td>132</td>
<td>$v = pc + pc_l$</td>
<td>148</td>
<td>$v = pc - pc_l$</td>
<td>164</td>
<td>$v = pc + pc_l$</td>
</tr>
<tr>
<td>133</td>
<td>$v = v_l + v$</td>
<td>149</td>
<td>$v = v_l - v$</td>
<td>165</td>
<td>$v = v_l + v$</td>
</tr>
<tr>
<td>134</td>
<td>$v = v_l + pc$</td>
<td>150</td>
<td>$v = v_l - pc$</td>
<td>166</td>
<td>$v = v_l + pc$</td>
</tr>
<tr>
<td>135</td>
<td>$v = pc_l + v$</td>
<td>151</td>
<td>$v = pc_l - v$</td>
<td>167</td>
<td>$v = pc_l + v$</td>
</tr>
<tr>
<td>136</td>
<td>$v = pc_l + pc$</td>
<td>152</td>
<td>$v = pc_l - pc$</td>
<td>168</td>
<td>$v = pc_l + pc$</td>
</tr>
<tr>
<td>137</td>
<td>$v = v + v_l$</td>
<td>153</td>
<td>$v = v - v_l$</td>
<td>169</td>
<td>$v = v + v_l$</td>
</tr>
<tr>
<td>138</td>
<td>$v = pc + v_l$</td>
<td>154</td>
<td>$v = pc - v_l$</td>
<td>170</td>
<td>$v = pc + v_l$</td>
</tr>
<tr>
<td>139</td>
<td>$v = v + pc_l$</td>
<td>155</td>
<td>$v = v - pc_l$</td>
<td>171</td>
<td>$v = v + pc_l$</td>
</tr>
<tr>
<td>140</td>
<td>$v = pc + pc_l$</td>
<td>156</td>
<td>$v = pc - pc_l$</td>
<td>172</td>
<td>$v = pc + pc_l$</td>
</tr>
<tr>
<td>141</td>
<td>$v = v_l + v$</td>
<td>157</td>
<td>$v = v_l - v$</td>
<td>173</td>
<td>$v = v_l + v$</td>
</tr>
<tr>
<td>142</td>
<td>$v = v_l + pc$</td>
<td>158</td>
<td>$v = v_l - pc$</td>
<td>174</td>
<td>$v = v_l + pc$</td>
</tr>
<tr>
<td>143</td>
<td>$v = pc_l + v$</td>
<td>159</td>
<td>$v = pc_l - v$</td>
<td>175</td>
<td>$v = pc_l + v$</td>
</tr>
<tr>
<td>144</td>
<td>$v = pc_l + pc$</td>
<td>160</td>
<td>$v = pc_l - pc$</td>
<td>176</td>
<td>$v = pc_l + pc$</td>
</tr>
</tbody>
</table>

The difference between 129-130 and 131-132 is that local delta is nonzero in 161-162.
<table>
<thead>
<tr>
<th>(v = v_l + d)</th>
<th>(v = v_l - d)</th>
<th>(v = v_l + d)</th>
<th>(v = v_l - d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>193</td>
<td>209</td>
<td>225</td>
<td>241</td>
</tr>
<tr>
<td>(v = pq + d)</td>
<td>(v = pq - d)</td>
<td>(v = pq + d)</td>
<td>(v = pq - d)</td>
</tr>
<tr>
<td>194</td>
<td>210</td>
<td>226</td>
<td>242</td>
</tr>
<tr>
<td>(v = d + v_l)</td>
<td>(v = d - v_l)</td>
<td>(v = d + v_l)</td>
<td>(v = d - v_l)</td>
</tr>
<tr>
<td>195</td>
<td>211</td>
<td>227</td>
<td>243</td>
</tr>
<tr>
<td>(v = d + pq)</td>
<td>(v = d - pq)</td>
<td>(v = d + pq)</td>
<td>(v = d - pq)</td>
</tr>
<tr>
<td>196</td>
<td>212</td>
<td>228</td>
<td>244</td>
</tr>
<tr>
<td>(pc = v_l + d)</td>
<td>(pc = v_l - d)</td>
<td>(pc = v_l + d)</td>
<td>(pc = v_l - d)</td>
</tr>
<tr>
<td>197</td>
<td>213</td>
<td>229</td>
<td>245</td>
</tr>
<tr>
<td>(pc = pq + d)</td>
<td>(pc = pq - d)</td>
<td>(pc = pq + d)</td>
<td>(pc = pq - d)</td>
</tr>
<tr>
<td>198</td>
<td>214</td>
<td>230</td>
<td>246</td>
</tr>
<tr>
<td>(pc = d + v_l)</td>
<td>(pc = d - v_l)</td>
<td>(pc = d + v_l)</td>
<td>(pc = d - v_l)</td>
</tr>
<tr>
<td>199</td>
<td>215</td>
<td>231</td>
<td>247</td>
</tr>
<tr>
<td>(pc = d + pq)</td>
<td>(pc = d - pq)</td>
<td>(pc = d + pq)</td>
<td>(pc = d - pq)</td>
</tr>
<tr>
<td>200</td>
<td>216</td>
<td>232</td>
<td>248</td>
</tr>
<tr>
<td>(d = v + v_l)</td>
<td>(d = v - v_l)</td>
<td>(d = v + v_l)</td>
<td>(d = v - v_l)</td>
</tr>
<tr>
<td>201</td>
<td>217</td>
<td>233</td>
<td>249</td>
</tr>
<tr>
<td>(d = pc + v_l)</td>
<td>(d = pc - v_l)</td>
<td>(d = pc + v_l)</td>
<td>(d = pc - v_l)</td>
</tr>
<tr>
<td>202</td>
<td>218</td>
<td>234</td>
<td>250</td>
</tr>
<tr>
<td>(d = v + pq)</td>
<td>(d = v - pq)</td>
<td>(d = v + pq)</td>
<td>(d = v - pq)</td>
</tr>
<tr>
<td>203</td>
<td>219</td>
<td>235</td>
<td>251</td>
</tr>
<tr>
<td>(d = pc + pq)</td>
<td>(d = pc - pq)</td>
<td>(d = pc + pq)</td>
<td>(d = pc - pq)</td>
</tr>
<tr>
<td>204</td>
<td>220</td>
<td>236</td>
<td>252</td>
</tr>
<tr>
<td>(d = v + v)</td>
<td>(d = v - v)</td>
<td>(d = v + v)</td>
<td>(d = v - v)</td>
</tr>
<tr>
<td>205</td>
<td>221</td>
<td>237</td>
<td>253</td>
</tr>
<tr>
<td>(d = v + pc)</td>
<td>(d = v - pc)</td>
<td>(d = v + pc)</td>
<td>(d = v - pc)</td>
</tr>
<tr>
<td>206</td>
<td>222</td>
<td>238</td>
<td>254</td>
</tr>
<tr>
<td>(d = pc + v)</td>
<td>(d = pc - v)</td>
<td>(d = pc + v)</td>
<td>(d = pc - v)</td>
</tr>
<tr>
<td>207</td>
<td>223</td>
<td>239</td>
<td>255</td>
</tr>
<tr>
<td>(d = pq + pc)</td>
<td>(d = pq - pc)</td>
<td>(d = pq + pc)</td>
<td>(d = pq - pc)</td>
</tr>
<tr>
<td>208</td>
<td>224</td>
<td>240</td>
<td>256</td>
</tr>
</tbody>
</table>

The difference between 202-244 and 226-296 is that \(\text{local-Gamma}\) is non-zero in 202-244.

<table>
<thead>
<tr>
<th>(v = v_l + v_l)</th>
<th>(v = v_l - v_l)</th>
<th>(v = v_l + v_l)</th>
<th>(v = v_l - v_l)</th>
</tr>
</thead>
<tbody>
<tr>
<td>257</td>
<td>269</td>
<td>281</td>
<td>293</td>
</tr>
<tr>
<td>(v = v_l + pc)</td>
<td>(v = v_l - pc)</td>
<td>(v = v_l + pc)</td>
<td>(v = v_l - pc)</td>
</tr>
<tr>
<td>258</td>
<td>270</td>
<td>282</td>
<td>294</td>
</tr>
<tr>
<td>(v = pq + v_l)</td>
<td>(v = pc + v_l)</td>
<td>(v = pq + v_l)</td>
<td>(v = pc + v_l)</td>
</tr>
<tr>
<td>259</td>
<td>271</td>
<td>283</td>
<td>295</td>
</tr>
<tr>
<td>(v = pq + pc)</td>
<td>(v = pq - pc)</td>
<td>(v = pq + pc)</td>
<td>(v = pq - pc)</td>
</tr>
<tr>
<td>260</td>
<td>272</td>
<td>284</td>
<td>296</td>
</tr>
<tr>
<td>(pc = v_l + v_l)</td>
<td>(pc = v_l - v_l)</td>
<td>(pc = v_l + v_l)</td>
<td>(pc = v_l - v_l)</td>
</tr>
<tr>
<td>261</td>
<td>273</td>
<td>285</td>
<td>297</td>
</tr>
<tr>
<td>(pc = pq + v_l)</td>
<td>(pc = pq - v_l)</td>
<td>(pc = pq + v_l)</td>
<td>(pc = pq - v_l)</td>
</tr>
<tr>
<td>262</td>
<td>274</td>
<td>286</td>
<td>298</td>
</tr>
<tr>
<td>(pc = pq + pc)</td>
<td>(pc = pq - pc)</td>
<td>(pc = pq + pc)</td>
<td>(pc = pq - pc)</td>
</tr>
<tr>
<td>263</td>
<td>275</td>
<td>288</td>
<td>300</td>
</tr>
<tr>
<td>(d = v_l + v_l)</td>
<td>(d = v_l - v_l)</td>
<td>(d = v_l + v_l)</td>
<td>(d = v_l - v_l)</td>
</tr>
<tr>
<td>265</td>
<td>277</td>
<td>290</td>
<td>301</td>
</tr>
<tr>
<td>(d = v_l + pc)</td>
<td>(d = v_l - pc)</td>
<td>(d = v_l + pc)</td>
<td>(d = v_l - pc)</td>
</tr>
<tr>
<td>266</td>
<td>278</td>
<td>291</td>
<td>302</td>
</tr>
<tr>
<td>(d = pq + v_l)</td>
<td>(d = pc + v_l)</td>
<td>(d = pq + v_l)</td>
<td>(d = pc + v_l)</td>
</tr>
<tr>
<td>267</td>
<td>279</td>
<td>293</td>
<td>303</td>
</tr>
<tr>
<td>(d = pq + pc)</td>
<td>(d = pq - pc)</td>
<td>(d = pq + pc)</td>
<td>(d = pq - pc)</td>
</tr>
<tr>
<td>268</td>
<td>280</td>
<td>292</td>
<td>304</td>
</tr>
</tbody>
</table>

The difference between 257-238 and 281-304 is that \(\text{local-Gamma}\) is non-zero in 257-304.
<table>
<thead>
<tr>
<th>Case</th>
<th>Rule</th>
<th>Case</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>353</td>
<td>$v := v_l + \eta$</td>
<td>365</td>
<td>$v == v_l + \eta$</td>
</tr>
<tr>
<td>354</td>
<td>$v := v_l + p\eta$</td>
<td>366</td>
<td>$v == v_l + p\eta$</td>
</tr>
<tr>
<td>355</td>
<td>$v := p\eta + v_l$</td>
<td>367</td>
<td>$v == p\eta + v_l$</td>
</tr>
<tr>
<td>356</td>
<td>$v := p\eta + p\eta$</td>
<td>368</td>
<td>$v == p\eta + p\eta$</td>
</tr>
<tr>
<td>357</td>
<td>$pc := v_l + v_l$</td>
<td>369</td>
<td>$pc == v_l + v_l$</td>
</tr>
<tr>
<td>358</td>
<td>$pc := v_l + p\eta$</td>
<td>370</td>
<td>$pc == v_l + p\eta$</td>
</tr>
<tr>
<td>359</td>
<td>$pc := p\eta + v_l$</td>
<td>371</td>
<td>$pc == p\eta + v_l$</td>
</tr>
<tr>
<td>360</td>
<td>$pc := p\eta + p\eta$</td>
<td>372</td>
<td>$pc == p\eta + p\eta$</td>
</tr>
<tr>
<td>361</td>
<td>$d := v_l + v_l$</td>
<td>373</td>
<td>$d == v_l + v_l$</td>
</tr>
<tr>
<td>362</td>
<td>$d := v_l + p\eta$</td>
<td>374</td>
<td>$d == v_l + p\eta$</td>
</tr>
<tr>
<td>363</td>
<td>$d := p\eta + v_l$</td>
<td>375</td>
<td>$d == p\eta + v_l$</td>
</tr>
<tr>
<td>364</td>
<td>$d := p\eta + p\eta$</td>
<td>376</td>
<td>$d == p\eta + p\eta$</td>
</tr>
</tbody>
</table>
Appendix B

Notes on using the CFL library

The code for the CFL can be found on the AP 3000 in the directory

```
~/tlh/Project/source/cfl
```

In this directory the code for the test harness can also be found. The code will compile and work as it stands. However, users should be aware that there will be some sections of the code that have been commented out and other sections are never used. If further work on the CFL is considered, it would be a good idea to gain a thorough understanding of the present code by reading it in conjunction with this report and then re-design the code to eliminate redundant parts, which at the moment only serve to confuse.

B.1 Programming with the CFL library

In order to use the library with an application, include the header file `cfl.hh` in the source for the application and link the file `cfl.o` with the object code for the application.

A `CFL_Manager` object is declared in the header file, and this is the communication manager used by the library at present. In order to initialise the communication manager with a communicator, enter the line

```
cm.init(any_communicator)
```

in the application program after the `MPI_Init(...)` statement. This initialises the library to work in the communicator passed to it (for example `MPI_COMM_WORLD`), and no further calls to the communication manager will be necessary in the application code.

To use a shared double, simply declare it as a `CFL_Double` and bear in mind the semantics discussed in chapter 3 when performing operations on the variable.