Investigating MPI streams as an alternative to halo exchange

Caoimhín Laoide-Kemp

August 20, 2015

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2015
Abstract

An alternative to the popular halo exchange algorithm was developed, utilising the concept of data streams in MPI. The code for this algorithm was developed from scratch and its performance was compared to a halo exchange code. Cases that were tested included the measurement of performance under strong and weak scaling, different problem sizes and durations, and different network speeds. The results indicate that the alternative algorithm consistently outperforms the halo exchange code, with up to 30x speedup being seen in some cases. However, it was also found that the usefulness of the algorithm was highly dependent on the type of problem being solved. Problems with long runtimes, small datasets, and frequent communication were where the most improvement was seen. Other problem types still saw some marginal improvement, but at the expense of a large amount of development work.
# Contents

1 Introduction ................................. 1
  1.1 Background ................................ 2
    1.1.1 Halo Exchange .......................... 2
    1.1.2 Streaming .............................. 3

2 Algorithm .................................. 5
  2.1 Problems with Halo Exchange .......... 5
  2.2 Halo Streaming Algorithm .............. 6
    2.2.1 Data Dependency ....................... 6
    2.2.2 Data Stream ............................ 7
    2.2.3 Primary Differences from Halo Exchange 8
  2.3 Messages .................................. 9
  2.4 Implementation ........................... 9
    2.4.1 Initialisation ......................... 10
    2.4.2 Extension ............................. 11
    2.4.3 Clean-Up .............................. 12
  2.5 Buffer Usage ............................. 13
  2.6 Parameters ............................... 14
  2.7 I/O ....................................... 15

3 Code Development ......................... 16
  3.1 First Implementation .................... 16
  3.2 Message Grouping ......................... 17
  3.3 Overlapping Sends and Receives .......... 18
  3.4 Verification ................................ 18
  3.5 Extension Phase ........................... 19
  3.6 Halo Exchange Code ........................ 19

4 Results and Analysis ..................... 21
  4.1 Results Details ........................... 21
  4.2 Morar Performance Tests ................. 22
    4.2.1 Small Work Buffer .................... 22
    4.2.2 Vampir Traces ......................... 23
    4.2.3 Larger Work Buffer .................... 25
    4.2.4 Message Size .......................... 26
4.3 ARCHER Performance Tests ........................................ 28
   4.3.1 Small Work Buffer ........................................... 28
   4.3.2 Single Core per Node ........................................ 30
   4.3.3 Strong Scaling .............................................. 31
   4.3.4 Huge Problem Size .......................................... 32
   4.3.5 Message Size .................................................. 35
   4.3.6 Halo Depth .................................................... 37

5 Conclusions & Final Analysis ...................................... 38

6 Future Work .......................................................... 41
   6.1 Multidimensional Cases ........................................ 41
   6.2 Extension of Functionality ..................................... 42
   6.3 Circular Buffer .................................................. 43
List of Figures

2.1 A halo exchange with a 3-point stencil. Green arrows represent transfer of halo data, while red arrows show the data used to calculate various portions of the next iteration ......................................................... 6
2.2 Computation of data without communication ......................................................... 7
2.3 Full computation using streaming model ................................................................. 8
2.4 Initial calculations. The process starts with the bottom row and begins to calculate the centres of the later iterations. ................................................................. 10
2.5 The process receives data (shown in red) from its neighbour. The calculations (shown in black) dependent on this new data are then performed until they have caught up with the other calculations. ................................................................. 11
2.6 Calculation continues in this manner until the full triangle is complete. ................. 11
2.7 The process receives data (shown in red) from its neighbour. All calculations (shown in black) dependent on this new data are then performed. This is in contrast to Figure 2.5, where calculations stop once they have caught up. ................. 12
2.8 Finally, the process receives the remainder of the data (shown in red) from its neighbour. All remaining calculations (shown in black) are completed. The process is then left with the top row of data shown in the diagram. ................. 13

4.1 Weak scaling on Morar. Work Buffer: 100 doubles per process. Iterations: 1,000,000. Message Size: 25 iterations. ................................................................. 22
4.2 Vampir visualisation showing the communication pattern of the halo streaming code. Red areas represent communication functions, green areas represent calculation. ................................................................. 23
4.3 Vampir visualisation showing the communication pattern of the halo exchange code. Red areas represent communication functions, green areas represent calculation. ................................................................. 24
4.4 Weak scaling on Morar. Work Buffer: 100,000 doubles per process. Iterations: 10,000. Message Size: 50 iterations. ................................................................. 25
4.5 Effect of message size on performance. Work Buffer: 100 doubles per process. Iterations: 1,000,000. ................................................................. 26
4.6 Weak scaling across nodes on ARCHER. Work Buffer: 100 doubles per process. Iterations: 1,000,000. Message Size: 50 iterations. ................................................................. 28
4.7 Weak scaling on ARCHER with large numbers of cores. Work Buffer: 100 doubles per process. Iterations: 1,000,000. Message Size: 50 iterations. ................. 29
4.8 Weak scaling with only one core used per node. Work Buffer: 100 doubles per process. Iterations: 1,000,000. Message Size: 50 iterations.

4.9 Strong scaling on ARCHER. Work Buffer: 1,638,400 doubles. Iterations: 10,000. Message Size: 50 iterations.

4.10 Speedup of a huge problem on ARCHER. Work Buffer: 983,040,000 doubles. Iterations: 100,000. Message Size: Variable.


Acknowledgements

I would like to thank Anna for all her hard work proofreading and keeping me sane, my parents for all the encouragement I could ever need, and of course my supervisor Dr. Daniel Holmes for the consistently helpful advice and for providing such an interesting project.
Chapter 1

Introduction

One of the most significant challenges faced by parallel computing is that of efficient communication between processors[21]. The advance towards exascale computing is resulting in much greater numbers of processors in the most powerful machines. This in turn increases the complexity and cost of constructing an efficient communications network. This trend is likely to continue for the foreseeable future, and will form a significant barrier in the search for ever-increasing performance[23].

In order to combat this, improvement must be made in all areas of processor communication[14], including:

- The performance of the physical interconnect
- The topology of the network
- Communication patterns

This project is primarily concerned with this last point. It aims to develop an alternative stream-based communication pattern using MPI[5]. The viability of this pattern is investigated, comparing it in terms of performance and flexibility to the widely-used halo exchange pattern.

The remainder of Chapter 1 presents some background information to the project, focusing in particular on the functions of halo exchange and defining what is meant by the term “streaming”. Chapter 2 describes the proposed streaming algorithm in detail, both in general concept and the specific implementation used in this project. Chapter 3 describes the process involved in developing the code for the algorithm, as well as the halo exchange code to which it was compared. Chapter 4 contains all the testing results, as well as an analysis thereof. Chapter 5 is a final summary of the project, including a discussion of the pros and cons of the streaming algorithm. Finally, Chapter 6 is a discussion of several topics that could be investigated as part of further research into this algorithm.
1.1 Background

1.1.1 Halo Exchange

Numerical linear algebra is a cornerstone of modern scientific computation[15]. It is the study of algorithms used to perform various linear algebra computations. This is necessary for determining numerical solutions to the differential equations that are common in scientific models.

Many methods are used to numerically solve linear algebra problems, including Gaussian Elimination[24], Krylov Subspace[9] and Jacobi Iterative[10]. These methods can be implemented using a variety of libraries, including LINPACK[3] and PETSc[6]. The performance and consistency of these libraries is of great importance in the field of HPC, as they are used as metrics for measuring the performance of large parallel machines, most notably in the Top500 list[7].

These methods act on large sets of data. Elements of these datasets are defined chiefly by two things: a value (or values) defining their state, as well as their neighbours within the global dataset. In general, each step of a numerical linear algebra calculation requires information from both the current element, as well as its neighbours. The pattern of element that is required is known as a stencil[20].

When dealing with problems that require a stencil, one commonly used parallelisation technique is domain decomposition[22]. Domain decomposition refers to the process of dividing the domain of the problem into smaller subsections. These subsections are then distributed across the hardware, with each subsection being dealt with by a different process. When performing calculations on any given data element, the process in question must have access to the data in all the cells within that element’s stencil. For data near the centre of the subsection this is not a problem, as all the necessary data will be located on the same process. However, for data on or near the subsection’s boundary, some of the required data will be located on other processes. It is clear to see that a communication pattern of some sort is required to transfer the necessary information when needed.

One commonly used communication pattern for domain decomposition problems is halo exchange[11]. Each process stores its boundary data in halo regions. At every timestep these regions are exchanged with neighbouring processes so that every process has access to the correct information. The processes then use their newly acquired information to update the halo regions for the next timestep.

Since halo data is exchanged at every step, a loose synchronisation comes into effect across the entire system. No process can be more than one step ahead of or behind its neighbours. Because of this, the algorithm can be modelled as a bulk synchronous parallel (BSP) algorithm[12]. In this model, each superstep consists of local computation, global communication, and barrier synchronisation. The communication period can also be used to perform global reductions of the data. This can be useful for tracking the global state of the system. The communication and calculation phases are often overlapped by using asynchronous communications to transfer the
Due to the barrier synchronisations, the time taken for a single superstep to complete is equal to the time taken by the slowest process. Because of this, delays to a single process propagate to its neighbours. This can cause a chain reaction and quickly result in all processes being delayed. If multiple processes are delayed at different points during the runtime, their delays tend to accumulate instead of overlapping. There are many sources that could cause these delays, including interrupts from the operating system or other programs competing for resources. Most of these effects do not occur on a large scale if the computing resources are well-managed, but system noise, even in small quantities, can cause dramatic decreases in performance[16]. This problem is likely to become worse with the arrival of exascale machines, as the chance of a process being delayed increases with the number of processors involved.

1.1.2 Streaming

A data stream is a communications concept referring to the manner in which data is transferred between two points across a network[8]. Instead of communicating in discrete messages at discrete moments in time (e.g. as is seen in halo exchange), there is a constant flow of information between the two endpoints. When the information becomes available to the sending process it is immediately inserted into the data stream. At this point, the sending process is free to move on with its work, without waiting to ensure that the receiving process has received the data. Conversely, the receiving process continually attempts to retrieve data from the stream whenever it is needed. In the ideal case, the required data will have already been sent into the stream whenever the receiving process makes a request for data.

While this project proposes a new software streaming algorithm, the concept of streaming data has long existed in hardware. Systolic architectures are one such example. Systolic architectures are characterised by their “assembly-line” nature. Data is passed from memory, through an array of processing elements, before being passed back to memory. Each processing element executes a fixed set of tasks on the data as it passes through. Systolic architectures are in contrast with von Neumann architectures, where the data remains localised on a processing element, to which instructions are passed to specify how the data should be manipulated.

The benefits and drawbacks of systolic architectures are well known[18]. One major advantage is the fact that systolic architectures reuse data many times for a single memory access. Von Neumann machines often have major bottleneck problems due to the expense involved in accessing memory. However, in spite of this, systolic architectures have fallen out of common usage. A primary reason for this is the fact that a systolic architecture must be designed with a specific problem in mind if it is to achieve the maximum possible performance. This can quickly become extremely expensive if there are a large number of distinct problems to be analysed. Modern HPC favours computers that can perform well for the wide variety of computational tasks that exist, so systolic architectures are not a good fit.

This project approaches the field of domain decomposition problems in a similar manner to
systolic architectures. The major similarity involved is that data is streamed between processing elements, each of which acts independently on its local dataset. The algorithm is also specialised to the specific task of optimising the time spent on communication in domain decomposition problem with a stencil. However, a significant difference is the fact that the algorithm is entirely software-based, without requiring any specific hardware. This avoids the problem associated with systolic architectures regarding over-specialisation. Software is much easier and cheaper to modify and reuse, so being specific to a single class of problems is not a major downside.

Another approach that has been taken to software-based data streaming was a proposal to introduce an extension to OpenMP3.0[19]. This extension would allow programmers to arrange for data to be streamed between parallel tasks. This was done by adding two optional clauses to the OpenMP task construct: input and output. These clauses would take a list of items as arguments, with each item describing data to be streamed and the behaviour of the stream. The paper gives several examples that demonstrate how this syntax can be used to parallelise several common OpenMP patterns, such as pipelines and filters. It also shows in detail how an FFT algorithm might be implemented using this model. Several performance results are quoted in the paper, with the most successful tests showing a speedup of 18.8 over standard methods.

This example shows the potential performance benefits that can be obtained by using streaming models. This project attempted a similar task, in that it combined the use of streams with already existent parallelisation methods. However, this project uses functionality that is already available within the MPI library to achieve its goal, whereas the cited example proposes an extension to OpenMP. Additionally OpenMP is traditionally suited for parallelisation on smaller machines, whereas this project will target improvements on as large a scale as possible.
Chapter 2

Algorithm

This chapter describes the basis of the proposed streaming algorithm. Section 2.1 discusses some of the problems with halo exchange that this algorithm aims to solve. Section 2.2 describes how the algorithm works, as well as noting the ways in which it differs from halo exchange. Section 2.3 discusses how the communications for this model are implemented using MPI. Section 2.4 gives a detailed account of how the algorithm was implemented in code. Section 2.5 explains how buffers are used in the code to achieve efficient data storage. Section 2.6 demonstrates how the algorithm can be tuned to optimise for specific problems. Section 2.7 describes how using this algorithm changes the approach to I/O.

2.1 Problems with Halo Exchange

Although halo exchange is extremely common in domain decomposition, the communication pattern has several problems that prevent it from being an ideal solution. As can be seen in Figure 2.1, each process must communicate with each of its neighbours at every timestep. This means that the performance of any given process is strongly dependent on the performance of its neighbours. This can result in delays to a single process quickly propagating across the system, bringing the entire program to a halt in extreme cases. These delays become more and more likely to occur as the number of processes increases.
Additionally, the halo exchange pattern does not easily allow for halo data from multiple iterations to be grouped together in a single message. Since the halo data from iteration $n$ is required before iteration $n+1$ can be completed, the preceding message must complete before the halo data from iteration $n+1$ can be sent. This means that each set of halo data will have its own separate communication overhead associated with it. This results in the performance of halo exchange programs dropping off significantly when measuring strong scaling. As the amount of calculation per process decreases, the communication overheads begin to represent the majority of the execution time. These overheads cannot be reduced by adding more processes; in fact the opposite is often the case. One way around this is to increase the depth of the halo region that is transferred. This reduces the number of communications that must be made, but this comes with its own disadvantages. These are discussed in more detail in Section 3.6.

2.2 Halo Streaming Algorithm

The primary goal of the alternative algorithm is to reduce (or remove) the problems described above. In other words, the performance of any single process should not be closely tied to the performance of any other, and the communication overheads should be reduced. The algorithm combines efficient exploitation of data dependencies with the concept of a data stream in order to achieve this.

2.2.1 Data Dependency

Communication between processes is necessary in domain decomposition problems because some of the calculations involved depend on data that is located on a different process. However, there is often a large amount of calculation that can be done using only the data on a single process. Taking a one-dimensional problem with a 3-point stencil as an example\(^1\), the process in question can update every point in its domain, apart from one at either end. This fact

\(^1\)A one-dimensional problem with a 3-point stencil is used for all examples and original results in this document.
is often used in halo exchange programs by overlapping this calculation of the central points with the communication of halo data.

At this point halo exchange codes usually wait to receive their neighbours’ halo data before continuing with the next iteration. However, as shown in Figure 2.2, it is possible to continue calculating chunks of data for many future iterations. In the example shown, each process can calculate a triangular area of data without needing to receive any data from its neighbours. However, none of the remaining data points can be calculated by any one process, as they depend on information from at least two processes.

![Figure 2.2: Computation of data without communication](image)

### 2.2.2 Data Stream

In contrast to halo exchange, where messages are sent and received periodically, the alternative communication pattern will utilise a data stream. In other words, the sending process will insert data into a communication stream whenever it becomes available, while the receiving process will retrieve data whenever it is needed. Data will be retrieved from the stream on a first in first out basis.

Figure 2.3 illustrates how the communication pattern functions. Each process begins by calculating any values that do not require communication (as shown in Figure 2.2). The central portion of the triangle can be discarded, as all data that is directly dependent on it has already been calculated. Data on the left hand side of the triangle can be written into a Send Buffer as
soon as it is calculated. Data points in the Send Buffer can be grouped together and sent in one large message.

Data from the right hand side of the triangle is retained, as it necessary for computing the missing data points between the triangles. This data can be obtained by receiving data from hand neighbour. As shown in Figure 2.3, this gives the process enough information to fill in the missing data. After each process has completed this procedure, the global data set will have been completely updated.

![Figure 2.3: Full computation using streaming model](image)

2.2.3 Primary Differences from Halo Exchange

As was discussed in Section 2.1, the messages in a halo exchange code must all be sent individually as each processor needs the data in the first message before it can generate the second one. The only way around this involves duplication of calculation, which is clearly undesirable. However, using the streaming model, the initial messages can all be generated from the process’s initial data. Because of this all the data (if necessary) can be sent in a single message, reducing communication overheads.

The dependency on neighbouring processes is greatly reduced. The result of the neighbouring process’s first iteration is not required until the entire triangular domain has been computed. This means (assuming that processes do not vary widely in their calculation speed) that when a
process attempts to receive some data from its stream the required information will have been
made available many iterations earlier. This should reduce the amount of time spent waiting
for other processes to catch up.

A halo exchange code has each process sending one data point to each of its two neighbours. For the streaming code, each process sends to only one of its neighbours, while receiving from the other. Additionally, each message consists of two data points instead of one. This can be clearly seen in Figure 2.3.

In a halo exchange code, each process will responsible for calculating a fixed domain of the problem throughout the program. However, when using the streaming model the domain associated with each process will shift in the opposite direction to the transfer of data. This is because each new packet of data that is received represents an area that is one data point further from the original boundary. This can again be seen in Figure 2.3. Because of this, halo streaming is more naturally suited to problems with periodic boundary conditions (although it can be adapted if this is not the case).

2.3 Messages

Although the communication pattern is modelled on a data stream, this functionality does not currently exist in MPI. As a replacement, the stream is simulated using a series of Point-to-Point communications. The processes each have circular Send and Receive Buffers, and use MPI's persistent communications functions to transfer the necessary data. If a section of either buffer needs to be reused, the process must wait until the send/receive that corresponds to that location has finished. Since persistent communications are non-blocking, this effectively acts as a data stream: initialising a persistent send corresponds to inserting data into the stream, while issuing a persistent receive equates to receiving the data at the head of the stream.

Persistent communications were chosen because they are theoretically the most efficient option when a message-passing routine is called multiple times with the same arguments[17]. Their use should ensure that the setup cost of creating an MPI request is performed only once. The same request can then be used each time the communication is issued. However, some versions of MPI do not implement persistent communications as intended. Persistent communications are often built on top of the implementation of the standard non-blocking communications, thereby removing any advantage to using them. Therefore, the use of persistent communications may actually decrease the performance of the code.

2.4 Implementation

Section 2.2.2 shows the basic concepts behind the halo steaming communication pattern. However, in practice, the implementation of the design is somewhat different. There are three
distinct phases to the calculation, which will referred to here as Initialisation, Extension, and Clean-Up.

2.4.1 Initialisation

In this phase, each process prioritises the completion of the initial triangular area of data. The phase ends when the peak of the triangle has been reached. The beginning of this phase is illustrated in Figure 2.4. The number of iterations that can be completed before reaching the peak is well-defined by the starting parameters. The number of data points that can be calculated in each subsequent iteration is reduced by the number of data points that are dependent on other processes. This means that each iteration has two calculations fewer to perform than the preceding one. If there are $N$ data points to begin with, then the number of iterations taken to reach the peak of the triangle will be $\frac{N}{2}$.

Figure 2.4: Initial calculations. The process starts with the bottom row and begins to calculate the centres of the later iterations.

As the phase progresses, the process will frequently check whether data has arrived from a neighbouring process. If it has, the process will calculate all data points that depend on the newly received data until it has caught up with the current iteration. This also has the effect of pushing the location of the peak further into the future, thereby extending the initialisation phase. This process is shown in Figures 2.5 and 2.6.
Figure 2.5: The process receives data (shown in red) from its neighbour. The calculations (shown in black) dependent on this new data are then performed until they have caught up with the other calculations.

Figure 2.6: Calculation continues in this manner until the full triangle is complete.

2.4.2 Extension

Once the triangle has been fully calculated, the process is wholly dependent on data belonging to its neighbouring process. At this point the process will block until it receives data. However, unless the neighbouring process has fallen many iteration behind, the data will have been sent many iterations previously. Once the data has been received, the process performs all possible calculations with the new information. This consists of creating a new strip of data along the side of the triangle. When the peak is reached the new data will be written into the Send Buffer.
and sent to the other neighbour. The process can then return to waiting for new data. This is shown in Figure 2.7.

It should be noted that receiving one iteration’s worth of data allows the process to push the peak of the triangle one iteration further. This is repeated until the iteration associated with the peak is equal to the final iteration. At this point the process will enter the Clean-Up phase.

Figure 2.7: The process receives data (shown in red) from its neighbour. All calculations (shown in black) dependent on this new data are then performed. This is in contrast to Figure 2.5, where calculations stop once they have caught up.

2.4.3 Clean-Up

Once the Clean-Up phase has been reached, the process will have no more data to send to its neighbour. All information necessary to finish calculating the final iteration will have already been sent. The process will receive information from its other neighbour one iteration at a time. This will then allow the next iteration to be calculated in its entirety. This continues until the final iteration has been completed, bringing the program to an end. The completion of this phase is illustrated in Figure 2.8.
Figure 2.8: Finally, the process receives the remainder of the data (shown in red) from its neighbour. All remaining calculations (shown in black) are completed. The process is then left with the top row of data shown in the diagram.

2.5 Buffer Usage

Each process has three primary buffers: Work, Send, and Receive.

The Work Buffer is where all calculations are performed. Any newly calculated values are stored in this buffer, and all calculations involve only data contained there. The size of this buffer is such that it can store all of the initial data, as well as one halo region. This extra region is used to store any newly received data.

The Send and Receive Buffers are both circular in nature. They are divided into blocks of data, each of which stores one outgoing or incoming message. The size of one of the blocks is equal to the size of the halo region multiplied by the number of iterations that are grouped together in a single message. The size of the entire buffer can be set to place a limit on the number of messages that can be outstanding at any one time. The smaller the buffer, the less memory is used unnecessarily, but the more each process is dependent on the performance of its neighbours.

It should be noted that each data element has three data elements from the next iteration that are dependent on it. For all except the first two elements on any given process, all three of those data elements are located on the same process. However, the first two elements’ dependents are split across the local process and the neighbouring process. Since these first two elements are copied into the Send Buffer before any calculation is done, the process does not need to ensure that all dependents are computed before the buffer is overwritten, merely the ones that are to be stored locally. It can clearly be seen that the first element has only one local element dependent on it. Therefore, once this term has been calculated, it can immediately overwrite the first element, which is no longer needed. The second element has two local dependents,
one of which has already been calculated and stored in the Work Buffer. The second dependent can therefore overwrite the second data element as soon as it is calculated. This trend of each data element being overwritten by the last of its dependents continues all throughout the buffer. The only elements that are not overwritten in this way are the final two. This is because the last of their dependents are located on the other neighbouring process. These elements are instead overwritten by the next chunk of data from the Receive Buffer.

This buffer usage has major consequences when using an explicit update method\(^2\). The traditional way of performing this kind of update is to use double-buffering. All updated elements are stored in a separate buffer to the originals. When all calculations have been performed, the positions of the original and updated buffers are switched, with the next set of updates being written into the original buffer. This uses twice as much memory as the method used in halo streaming.

### 2.6 Parameters

The halo streaming method has several parameters that can be tuned in order to optimise performance. This is in contrast to halo exchange, which has limited scope for adjusting the code to meet specific requirements.

Firstly, the size of the messages that are transferred between processes can be altered to reflect the size of the problem and the speed of the communication network. Using a small message size means that more messages in total must be sent. This results in each process spending a higher percentage of its time communicating, and a lower percentage calculating, which is undesirable. However, if the message size is too large, processes may become too tightly coupled. To illustrate this, take the example of each process having an initial domain of 100 data elements. If a message size of 50 is chosen then each process will compute 50 iterations before sending its data. However, with only 100 data elements, the initial triangle will be completed after 50 iterations. This means that no process will have any more calculations to perform by the time they send their first message. In the case of a perfect machine where all processes proceed in lockstep, this will result in all processes having to wait for their neighbour’s message to arrive. This defeats the purpose of using halo streaming.

The size of the Send and Receive Buffers can be altered to reflect the machine hardware. The size of these buffers define the number of persistent communications that can exist within the program. If this number is small, the communication handle will be reused more often. This can result in the propagation of delays if one process lags behind the others. However, if size is too large it can consume more memory than is necessary. This can cause unnecessary cache misses. The size of the Send and Receive Buffers do not necessarily have to be identical (or indeed related in any way). If a process is expected to lag behind its neighbours, then it should be more efficient to use a smaller Send Buffer (as outstanding sends should complete

\(^2\)An explicit method is one where the update is made using only data from the current state of the system. This is in contrast to implicit methods, where both the current state and updated state are used
more quickly) and a larger Receive Buffer (as incoming messages may build up). However, all results quoted in this report used an identical size for both buffers, as the target machine was assumed to be homogeneous.

2.7 I/O

In domain decomposition codes MPI-IO is often used to write the final data to disk[13]. To do this, each process specifies the area of the file that corresponds to its local data. All processes then write their data simultaneously to the same file, greatly reducing the amount of time spent on I/O. For halo exchange problems, this is a relatively straightforward task. Each process’ data corresponds to a contiguous chunk of memory on disk. This means that the mapping from the process’ local data to its position in the global dataset will be extremely simple.

However, when using halo streaming, the local domain of each process shifts in the opposite direction to the stream with each iteration. Because of this, it is necessary for every process to track its location within the global dataset. Additionally, this means that one of the processes will contain data from both the beginning and the end of the global dataset. Attempting to write this to disk naively results in data being written to the incorrect part of the file. This is because the data at the end of the local buffer corresponds to the location at the beginning of the global file, and vice versa. MPI-IO does not have enough information to realise this, and so the data is written to file in the order it is stored in the buffer.

In order to circumvent this problem, the data in the local buffer is shifted around into the correct order before being written to disk. This ensures that the MPI-IO can be conducted as usual. However, for very large arrays this shifting of data can be both time and memory intensive. Additionally, all of the work involved must be done by a single process. This means that the entire program effectively operates in serial for this portion of the I/O. This means that I/O operations can take considerably longer when using halo streaming over halo exchange.

It is worth noting however, that this increase in I/O time only occurs if it is necessary to have the global dataset begin at a fixed point. Since the code deals with systems with periodic boundary conditions, it may be acceptable to have the dataset begin at any point. If this is the case, the data can be written to file in the order it appears on the processes.
Chapter 3

Code Development

This chapter gives a summary of how the state of the code changed during the development phase. Section 3.1 describes the initial attempt at creating a working implementation of the algorithm. Section 3.2 documents several changes that were made to manner in which the code performed communications. Section 3.3 discusses the changes made to allow the code to adapt its priorities to the data that was available. Section 3.4 is a summary of how the output of the code was verified to be correct. Section 3.5 concludes the development of the halo streaming code by describing the extension to a full streaming model. Lastly, Section 3.6 gives a brief description of the halo exchange code.

3.1 First Implementation

The initial goal was to create a proof of concept. That is, to write a code that executed the basic streaming model. This was done to ensure that creating a data stream was possible within MPI, and to show that the halo streaming model would produce the same final output as halo exchange. No emphasis was put on optimisation or efficiency.

The first implementation contained only the minimum possible functionality to create a streaming data pattern. The program consisted of a shorter version of the full model. The initial triangle was calculated, sending a separate message at the beginning of each iteration. No attempt to receive messages was made until the entire triangle was complete. Once this stage had completed, the program then began to complete the missing inverted triangle, without entering the Extension phase.

For communication, persistent sends were used, along with standard receives. It was necessary for the sends to be non-blocking in order to avoid deadlock. Since no process attempts to receive a message before having sent all its data, using blocking sends would result in all processes coming to a halt waiting for their neighbour to post a receive. There is no such restriction on the receives, as the data does not need to be received until it is needed for calculations. In
addition, since only one message was received and processed at a time, there was no need for
a Receive Buffer. The message was simply received into the end of the Work Buffer.

This structure placed strict restrictions on the functionality of the code. The code did not alter
its order of calculations based on the availability of information, instead progressing in a linear
fashion. Additionally, omitting the Extension Phase places a limit on the number of iterations
that can be performed by the code. Since the process of filling in missing data does not extend
the calculation further into future iterations, the code will go no further than the final iteration
of the initial triangle. This is obviously not ideal as it means that the code is restricted to
problems where the initial domain size of each process is large enough to allow them to reach
the target iteration.

The actual updates performed were simply averaging each data element with its nearest neigh-
bours. This was chosen as an example of a 3-point stencil to give an indication of the perfor-
mance that might be expected in applications using similar algorithms. This also meant that
the final output of the code would be predictable and easy to verify.

The validity of the final output was checked by generating a checksum from a reduction of the
global dataset. This allowed the results of the streaming algorithm to be quickly and easily
compared with those from the halo exchange code. The results from this version of the code
were promising, showing large increase in performance over the halo exchange code in certain
situations.

3.2 Message Grouping

The next step was to improve the flexibility and performance of the communication pattern.
The standard receives were all replaced with their non-blocking variants. This allowed the
communications to be completed more quickly, freeing up memory in the Send Buffer earlier.
This did not actually change the manner in which the code performed its functions, but it was
a necessary alteration that allowed the data stream to become more flexible to future changes.

Introducing non-blocking receives meant that each process would always have multiple incom-
ing messages outstanding. These cannot be allowed to write to the same location (the end of
the Work Buffer, as was the case originally), as they would overwrite data from previous mes-
sages before it had been used. Because of this, each receive was directed to a different part of a
Receive Buffer. Information was then copied from this buffer to the Work Buffer when it was
required for calculations.

The communication frequency was changed so that each process completed a chunk of iter-
ations between communications. This chunk was of a fixed size, and determined the size of
the messages that were transferred between processes. Data from the same chunk of iterations
were grouped together in the Send Buffer and transmitted in one large message. Similarly,
data was received in chunks of the same size. This allowed the code to be adjusted in order to
minimise the time spent on communication.
3.3 Overlapping Sends and Receives

At this point, the code still processed all of the calculations corresponding to the initial data before beginning to deal with the received data. While this may function acceptably for a short problem, this is clearly infeasible for situations where many iterations must be calculated. This would result in the incoming data building up until memory capacity is reached.

To fix this problem, it was necessary to allow the code to both send and receive data when appropriate. To do this, the program used MPI_Test after every chunk of iterations to check if any new data had arrived. If no new data was available then the program continued to calculating and sending the next chunk of data. However, if new data had been received it was then used to calculate the missing data from the subsequent chunk of iterations. If the received data corresponded to a block of \( n \) iterations starting from iteration \( N \), then it would be used to complete iterations \( N + 1 \) to \( N + n + 1 \).

This change meant that the code no longer followed a deterministic path throughout its duration. This allowed the program to be more flexible in its execution, making best use of the available information in order to maximise the amount of data that is generated to send to the next process, as well as ensuring that information from the previous process is dealt with as soon as possible. However, the fact that the code’s performance was no longer deterministic made the task of verifying the accuracy of the output more difficult. For this reason, it was decided to alter the method used to ensure correctness.

3.4 Verification

The initial method of verifying the correctness of the output was to perform a global reduction of the data. This was then used to generate a checksum that could be compared to a value created the same way using the halo exchange data. This provided a quick and easy way to ensure that the data matched. However, it became clear that this method gave no qualitative information regarding the nature of any errors that occurred. In order to rectify this, the verification process was changed to writing the global dataset to two files: one for halo streaming and the other for halo exchange. The files were then compared line by line to ensure that the datasets matched.

While this method does take much longer than the checksum technique, as well as using vastly more memory, it has a number of advantages. The first is that comparing the datasets this way shows exactly which parts of the datasets differ, as well as how they are different. This information can be used to identify and correct sources of error. This is considerably more useful than the checksum, which simply gives a binary Pass/Fail notification. Secondly, the use of this method provided motivation to investigate the problem of how best to write the halo streaming dataset to file. This is discussed more fully in Section 2.7.
3.5 Extension Phase

The final major change to the code was to remove the limit on the number of iterations that could be calculated. This was done by adding the Extension Phase discussed in Section 2.4.2.

The primary change that this caused to the performance of the code was the manner in which the processes processed incoming data. Originally, upon receiving data from iteration $N$, the code would use the information to calculate all missing data from iterations $N + 1$. This method works well when at least part of the last iteration can be calculated solely from the initial data, but not if the code must carry on beyond this point. This can be seen from the fact that once the initial triangle has been completed, the code cannot progress to future iterations until at least part of the peak iteration has been calculated. Prioritising the full completion of the earliest iteration before moving on means that the code cannot continue past the peak of the triangle until all of the missing information has been calculated. This results in a long period of time in which no new send data is being generated. This effectively acts like a pseudo barrier in the code: no process can progress past this point until its neighbour has caught up. This clearly defeats the purpose of halo streaming, which aims to remove the dependence on the performance of neighbours.

To solve this problem, whenever a new piece of data is received the program calculates all possible calculations that depend on that data. These calculations continue until they have caught up with the data that has already been calculated. This then allows the process to generate a new package of information from the next iteration, which can be sent to the neighbouring process. Each process can continue this procedure indefinitely, until the target number of iterations has been reached.

3.6 Halo Exchange Code

In order to acquire relevant results, the aim was to construct the halo exchange code as it would appear in most HPC applications. To do this, the code was kept extremely simple, with no major problem-specific optimisations. Additionally, it was necessary to ensure fairness of comparisons with the halo streaming model. To make sure of this, it was decided that no optimisation could be made to one method without also applying it to the other if possible. Optimisations that were only relevant to one of the methods were considered an intrinsic advantage of that method. As an example of this, improving the halo streaming code by grouping messages together was considered reasonable, as the ability to do this is an advantage to using that method. However, vectorising one of the codes to take better advantage of the target architecture was considered an unfair comparison.

The halo exchange code communicated entirely using non-blocking communications. At the beginning of each iteration the each process issues two sends and two receives, one of each for each of its neighbours. Then, they update all cells that can be done without the use of halo data (i.e. all cells except the ones on each end). These updates are written into a different
buffer to avoid overwriting the original data. The processes then wait for their receives to complete, using them to update the end cells. At this point, the processes wait until their sends have completed, to ensure that the data has been received before the data is overwritten. The two arrays are then switched, so that the next set of calculations will be performed using the updated data.

One non-standard functionality that was temporarily added to the halo exchange code was the ability to send and receive halos of different depths. The goal of doing this is to reduce the frequency with which the processes need to communicate. For example, if each message consists of ten data elements instead of one, communication is only required once every ten iterations. However, the trade-off for this is that each process has more calculations to perform, since some of the calculations on cells in the halo region have to be duplicated across two processes.

In order to allow different halo depths, the manner in which data was transferred had to be changed. Since the buffers are overwritten several times between each communication, it is no longer sufficient to send data directly from the Work Buffer. Instead, the halo region must be copied into a separate communication buffer before being sent. As with the original version, the code performs all possible calculations involving the original data. The processes then wait for their neighbours’ data to arrive, before using it to complete the remaining calculations. Then, before the next iteration has begun, the processes wait to ensure that their sends have completed before the Send Buffer is overwritten by the next set of halo data.
Chapter 4

Results and Analysis

4.1 Results Details

The majority of the results presented in this report were obtained by running the code on ARCHER. Full hardware details of the cluster are available[1], but some of the important ones will be listed here. ARCHER is divided into nodes, each of which contains two 2.7 GHz 12-core processors. Each processor is a single NUMA region, with the two processors inside a node connected by two QuickPath Interconnect links. The nodes are connected using the Dragonfly topology.

The remaining results were obtained on EPCC’s Morar[4]. Morar was used primarily at the beginning of the testing period. The machine gives less consistent performance than ARCHER, but queue times are considerably shorter. This made it ideal for obtaining the initial performance results.

The execution time was measured using the MPI_Wtime function. The timing recorded is the time taken for the program to complete all the update calculations. In other words, the following were not included in the timing results:

- Initialisation of buffers
- Generation of communicators and persistent communications
- I/O

In order to achieve accurate results, the models were run 11 times, with the average of the last ten taken as the final result. The first run was discarded as it was found to consistently be an outlier in the distribution. This was most likely due to the fact that the Work Buffer suffered from a compulsory cache miss during the first iteration of calculation. All 22 runs (11 each for halo streaming and halo exchange) were run in the same executable file, meaning that the same processors in the same layout were used in each case.
4.2 Morar Performance Tests

Before beginning to test on ARCHER, it was decided to run some smaller tests on Morar first. These tests were used to verify the correctness of the code, as well as determining whether there was any evidence of increased performance. Both the number of processes and the length of the trials were kept short. The former due to machine constraints, and the latter to ensure that the preliminary results could be accumulated quickly.

4.2.1 Small Work Buffer

The first scenario tested was with an extremely small Work Buffer (100 floating point doubles per process) and 1,000,000 iterations of calculation. This was done so that there would be a large amount of communication relative to the calculation. It was expected that the halo streaming code should outperform the halo exchange equivalent, due to the fact that the halo streaming model relies less on the speed of communication to achieve good performance.

![Figure 4.1: Weak scaling on Morar. Work Buffer: 100 doubles per process. Iterations: 1,000,000. Message Size: 25 iterations.](image-url)
Figure 4.1 shows the results of this test. As can clearly be seen, the halo streaming code outperforms the halo exchange at every stage. Not only that, but the gap between the two increases as the number of cores gets larger. This is as expected, since increasing the number of cores increases the chance of incurring a delay in one of them. As previously discussed, this was expected to have a large effect on the halo exchange, but not on the halo streaming.

Since the number of calculations performed by each process was constant, one would expect the execution time to be the same on a perfect machine, regardless of the number of processes. As can be seen, this is clearly not the case for the exchange code, but the execution time for the streaming code is also seen to increase slightly. Additionally, timing results for the lower numbers of cores were quite consistent, but became very erratic when extended to more cores. For example, the range of the halo exchange results with two cores was 5.4-6.0 seconds. This changed to 19-55 seconds when 64 cores were used. This can be explained by noting that Morar does not quarantine its jobs in the same way that some supercomputers do. During periods of high demand, performance can suffer as jobs compete for resources. This was a motivator for using ARCHER for some of the future timing results, as its system ensures that all jobs will have the uninterrupted access to their resources.

### 4.2.2 Vampir Traces

![Vampir visualisation showing the communication pattern of the halo streaming code. Red areas represent communication functions, green areas represent calculation.](image1)
In order to verify that the results in the previous section were correct, the code was profiled using Vampir to investigate the reason behind the disparity between the two methods. Figures 4.2 and 4.3 show a section of the visualisation of this profiling. As can clearly be seen, the halo streaming code spends most of its time performing calculations. Any communication functions complete almost instantaneously.

In contrast, the halo exchange code spends a much higher percentage of its time waiting for communications to complete. This causes the program to take much longer to complete its calculations. One other factor to note is the frequency of communication. For these tests, the halo streaming code packaged its messages into groups of 25 iterations each. This means that each process communicates only twice (one send and one receive) every 25 iterations, while the halo exchange processes must communicate 100 times (two sends and two receives every iteration). This results in much less communication overhead and greatly contributes to the reduction in execution time.
4.2.3 Larger Work Buffer

The tests were redone with a larger Work Buffer (100,000 floating point doubles) and a smaller number of iterations (10,000). The results are shown in Figure 4.4.

![Figure 4.4: Weak scaling on Morar. Work Buffer: 100,000 doubles per process. Iterations: 10,000. Message Size: 50 iterations.](image)

In this case, the streaming model still outperforms the halo exchange, but the gap is much smaller. At low core counts the difference in execution time is very small. It is not until larger numbers are used that the halo streaming becomes noticeably better. In this scenario, the ratio of communication to calculation is smaller, which means that the performance of the halo exchange is not impacted as much as with the smaller dataset. Additionally, with the smaller number of iterations, the halo streaming code never exits the Initialisation phase. This is because the initial buffer contains enough data to ensure that the final iteration will be reached before the peak of the triangle. This means that each process will reach the point where data from their neighbour is required more quickly than if the trial were allowed to run for longer. This increases the dependency on the neighbouring processes, thereby decreasing the ability of the code to overlap communication with calculation.
4.2.4 Message Size

![Graph showing the effect of message size on performance.](image)

Figure 4.5: Effect of message size on performance. Work Buffer: 100 doubles per process. Iterations: 1,000,000.

The effect of the message size on the halo streaming performance was also investigated on Morar. This was done to provide an insight into the importance of choosing the correct message size for the job at hand. Again, the Work Buffer with 100 floating point doubles was used. The trials were run for message sizes varying from 1 to 50 iterations worth of data. Note that 50 is the largest message size possible for a Work Buffer with 100 data elements, as the initial data only contains enough information to perform 50 iterations of calculation.

As can be seen in Figure 4.5, the performance of the code generally improves as the message size increases. The only exception to this is when using 64 cores. In this case the performance is slightly better when using a message size of 25 than it is with 50.

These results fit with the expected performance of the code. In general, larger message sizes should perform better, since the communications overhead is reduced. As was shown in the Vampir visualisations, one of the major advantages of halo streaming over halo exchange is the relatively low frequency with which communications occur. Accentuating this aspect of
the code gives even more improvement. However, in this specific case, a message size of 50
represents the entire amount of work that a process can complete without communication. This
means that by the time a process sends its first message, it will have no more work that it can
complete. Since each process is in the same situation, some processes will have to wait until
their neighbours finish before they can continue their calculations. This can cause delays to
propagate in the same way as halo exchange. However, there are still advantages over halo
exchange, namely that this loose synchronisation only occurs once every 50 iterations, and
each process is only tied to one neighbour instead of two. This effect becomes apparent when
using 64 cores, as the number of delays and interruptions increases.
4.3 ARCHER Performance Tests

The first task to be done on ARCHER was the replication of some of the tests that were done on Morar. This was done in order to ensure that the improved performance was not specific to a single machine, as well as to investigate any changes that the use of a more stable and powerful machine might bring about. It was expected that the results would be much more consistent than on Morar. This is because of the way ARCHER assigns cores to a job. When any core on a given node is used, the entire node is given over to that job. This means that the job can be isolated from the rest of the machine, preventing other jobs from interrupting its progress.

4.3.1 Small Work Buffer

The tests with the small Work Buffer of 100 floating point doubles were repeated. The size of the job was scaled in multiples of 24 cores, as this constitutes a full node on ARCHER.

![Figure 4.6: Weak scaling across nodes on ARCHER. Work Buffer: 100 doubles per process. Iterations: 1,000,000. Message Size: 50 iterations.](image-url)
As can be seen in Figure 4.6, the halo streaming code continues to outperform halo exchange. When using one node, the streaming is already four times faster than its exchange equivalent, with the difference between the two only increasing with the number of nodes. One major point of note is the fact that the both codes scale better on ARCHER than they did on Morar. This is likely to be due to the way ARCHER isolates its jobs, as discussed above. However, the streaming code continues to scale much better. By the time eight nodes are used, the parallel efficiency of the exchange code has dropped to 50% (with respect to the performance on one node), while the streaming code remains >90%.

It can also be noted that there is a sharp increase in execution time for the halo exchange code between one and two nodes. This is because the communication speed within a node is much faster than the speed of the inter-node network. Cores that have to communicate with neighbours that are off-node have to wait longer to receive their data, as well as to receive confirmation that their send data has arrived. This causes delays in these processes that then spread throughout the network. However, the reduced communication speed does not impact the performance of the halo streaming code, as the data should still arrive before it is needed.

This test was then extended to up to 32,768 cores in order to discover how well the code coped with huge numbers of cores. It was again expected that the halo streaming would continue to outperform the halo exchange equivalent. The results are shown in Figure 4.7.

![Figure 4.7: Weak scaling on ARCHER with large numbers of cores. Work Buffer: 100 doubles per process. Iterations: 1,000,000. Message Size: 50 iterations.](image)
The performance of the halo streaming code is slightly worse with these larger numbers of cores. However, the execution time on 32,768 cores is still comparable to the lowest tested time on one node. The increase in time is approximately 30%. Compare this to the halo exchange times, where the increase for the same range is over 450%. (This is despite having a higher base time.) At 32,768 cores, the halo streaming model shows a performance over 20 times faster than that of halo exchange.

4.3.2 Single Core per Node

As was shown earlier, the performance of halo exchange decreases sharply when inter-node communication is present. This characteristic is not present in the results for halo streaming. In order to further investigate the effects of the network bandwidth on the performance of the code, the tests were repeated using only a single core per node. Doing this ensured that every single communication would use the inter-node network, instead of the links inside the node. This slows down the communication speed dramatically.

![Figure 4.8: Weak scaling with only one core used per node. Work Buffer: 100 doubles per process. Iterations: 1,000,000. Message Size: 50 iterations.](image)
In this case, the halo streaming outperforms the halo exchange by an even larger factor than usual. The halo exchange code scales extremely poorly, with the program taking nearly four times longer to run on 512 cores than on two. The halo streaming code performs much better. When only two cores are used, the performance is already nearly ten times faster than with halo exchange. This increases to over 30 times faster when using 512 cores. While the streaming code scales much better than halo exchange in this case, it should be noted that it does not scale perfectly. This indicates that the decrease in communication speed does have an effect on its performance, albeit much less than is seen with halo exchange.

Another aspect of these results that should be noted is the inconsistency of the halo exchange results. As can be seen from the error bars, the timing results for the halo exchange code varied quite strongly. This test was run with a number of different versions of the code before the final results were obtained. These results varied even more strongly, possibly due to different network conditions at the time. It should also be noted that this does not affect the halo streaming code in any noticeable way.

### 4.3.3 Strong Scaling

![Strong scaling graph](image)

**Figure 4.9:** Strong scaling on ARCHER. Work Buffer: 1,638,400 doubles. Iterations: 10,000. Message Size: 50 iterations

Having investigated how the models performed under weak scaling, the next logical step was
to investigate strong scaling. The total problem size was chosen so that the largest number of cores used (16,384) would each have a Work Buffer of 100 floating point doubles, as was the case in the weak scaling. As the total problem size (1,638,400 floating point doubles) was several orders of magnitude larger than with the weak scaling, the number of iterations that were computed was reduced in order to satisfy time constraints.

As can be seen in Figure 4.9, halo streaming is still the quicker of the two methods. However, in this case the difference between the two is much smaller until extremely large numbers of cores are used. When the number of cores is small, halo exchange is not as strongly affected by delays, because they happen proportionally less often. Additionally, when each process has a large Work Buffer and (relatively) few iterations, the cores are restricted in the amount of calculation that can be performed without communication, and are therefore more closely tied to the performance of their neighbours. These factors combine to make the difference between the two models minimal. However at large numbers of cores, we again see the streaming model outperforming halo exchange.

### 4.3.4 Huge Problem Size

Up until this point, all tests had involved small Work Buffers. This was useful for generating results quickly, as well as for investigating certain aspects of the code. However, this also meant that the results were not applicable to many large-scale HPC applications. These applications are often extremely memory intensive, with each step of calculation taking a long amount of time. In order to obtain results that were relevant to these problems, a large fixed-size problem was chosen and used to compare the models.

For these tests, a Work Buffer of 983,040,000 floating point doubles was chosen. This corresponds to just under 8GB of memory, which means that the entire dataset can fit in half the memory of a single node on ARCHER. More than half of the total memory cannot be used, as the halo exchange code must double-buffer the data. The test was run for 100,000 iterations. This was large enough to ensure that the code had enough time to make relevant comparisons, even on the highest number of cores. Additionally, it was low enough that the results on low numbers of cores could still be obtained in a reasonable amount of time.

The primary aim of the tests was to determine the best possible performance for each method. Because of this, the parameters of the halo streaming code were tuned throughout the course of the tests in order to find the combination with the best performance. This tuning primarily consisted of increasing the size of the messages as the number of processes increased, in order to reduce the dependency on neighbours.
Since the one-process versions of the two different models are different, a separate serial code was made. This was done to allow the speedup of the two codes to be compared to the same base time. However, since the Work Buffer was so large, it was not possible to run the serial code for the complete test. Extrapolations indicated that running the full test would take nearly half a year. To circumvent this, the serial code was run for just ten iterations. The base time was then extrapolated from this.

![Speedup Graph](image)

**Figure 4.10**: Speedup of a huge problem on ARCHER. Work Buffer: 983,040,000 doubles. Iterations: 100,000. Message Size: Variable

There are several points of note in Figure 4.10. The first is that the halo streaming model again performs better than halo exchange. The performance is noticeably better at all core counts. Secondly, both models demonstrate superlinear speedup. Thirdly, each of the models has a point in its graph at which doubling the number of cores increases its performance by a factor of ten. However, this jump does not occur at the same number of cores in each case.

The consistent out-performance of the streaming code is just what would have been expected from all the previous test results. Since the buffer size per process is not particularly small, nor the number of iterations particularly large, the advantages over halo exchange are limited. This means that halo streaming does not drastically outperform halo exchange at all points in the graph, but still displays some benefits over the other model.
The superlinear scaling is most likely due to improved memory access time. When all the data is located on a single processor, most of the memory accesses will be from main memory. This is extremely slow, and is likely to affect performance much more than calculation or communication issues. However, when the data is split across multiple processors, each core may be able to store a larger portion of its data in cache, thereby reducing the amount of time taken to retrieve data from memory.

The sudden large increase in performance at certain core counts can be similarly attributed to cache effects. This occurs when each core’s Work Buffer(s) can fit into the cache. In this scenario, all memory reads are from the cache. Since cache reads are much faster than those from main memory, this has a significant impact on the performance of the code. As can be seen from the graph, the transition happens at 4,096 cores for halo streaming, but 8,192 cores for halo exchange. This is because halo streaming uses only one Work Buffer, whereas halo exchange uses two, requiring double the memory. This result fits with the architecture of the Ivy Bridge processors used in ARCHER[2]. At 4,096 cores, each core has a Work Buffer of 240,000 floating point doubles, or 1.92 MB. On these processors, 12 cores share an L3 cache, so the total amount of data that needs to be stored is 23.04 MB. This is small enough to fit in the 30 MB L3 cache. For halo exchange, twice the data is used, so this transition happens at 8,192 cores.

Figure 4.11: Relative Parallel Efficiency of the two models using the huge problem. Work Buffer: 983,040,000 doubles. Iterations: 100,000. Message Size: Variable
This can be seen more clearly in the Figure 4.11, which shows the relative parallel efficiency\(^1\) of each model when compared to the time taken at 128 cores. The plots for the two models are very similar, apart from the sharp increases occurring at different points.

This result shows an important benefit of the halo streaming algorithm that had not been a key factor in its design. Although both methods benefit somewhat equally from cache effects, the fact that the streaming model uses only one buffer means that these cache effects come into play at a lower number of cores than algorithms that use double-buffering. While this should not have any effect on how quickly the code can run given unlimited resources, it should reduce the quantity of resources required to achieve acceptable performance.

Although this feature benefits the halo streaming code greatly, it is not specific to that model. In other words, it should be possible to develop a code that gains the caching benefits of halo exchange without implementing the full communication model. This would be done by copying the first two data points to a separate buffer. Those points could then be overwritten in the Work Buffer in the same way as halo streaming. The communication could be performed using normal halo exchange (or otherwise). This could potentially be useful for an application where the more efficient use of memory would be beneficial, but the implementation of the full halo streaming model would be overly complex or time-consuming.

### 4.3.5 Message Size

Another result that became apparent during these trials was the sensitivity of the performance of the halo streaming code to the message size. For core counts up to 8,192 a message size of 50 was used, with all timing results being as expected. However, the initial results for 16,384 cores were nearly four times slower than was anticipated by extrapolation. It was unknown whether this was a problem with the halo streaming model in general, or if it could be fixed by tuning some of the parameters. As can be seen from Figure 4.12, using a smaller message size results in the performance returning to what would be expected. This indicates that choosing the correct message size is crucial to achieving an efficient halo streaming code. In this case, choosing a message size that is too large results in the code performing much worse than even halo exchange.

\(^1\)This quantity is calculated for \(N\) cores using the following formula: 

\[
E_N = \frac{T_{128} \times 128}{T_N \times N},
\]

where \(E_N\) and \(T_N\) are the relative parallel efficiency and time taken on \(N\) cores respectively. This gives an efficiency of 1 if doubling the number of cores halves the execution time.
The fact that the execution time spikes so dramatically at 50 iterations was not expected. The expectation was that performance would decrease steadily past the optimum size, but the results show a huge difference between message sizes of 25 and 50. There was not enough time to investigate this fully, but a number of possible solutions were discussed. The most likely is that the message size negatively affects cache usage. The sizes of the Send and Receive Buffers scale with the size of the messages. Therefore, if the message size is too large, it may cause data to be missing from the cache.
4.3.6 Halo Depth

Figure 4.13: Effect of halo depth on the performance of the huge problem. Work Buffer: 983,040,000 doubles. Iterations: 100,000. Cores: 512

The message size had a large effect on the performance of the halo streaming code, so it was decided to run the halo exchange code again with varying halo depths in order to see if it could match its performance. As can be seen from Figure 4.13, altering the halo depth did have some impact on the performance, but not nearly enough to make up the gap between the two models. There was not sufficient time to investigate if this was the case in all scenarios, but there is currently no reason to think otherwise.
Chapter 5

Conclusions & Final Analysis

The primary goal of the halo streaming algorithm is to improve on halo exchange by reducing the time spent on communication between processes. By making efficient use of data dependencies and data streams, the algorithm aims to overlap communication and calculation as much as possible, resulting in a more efficient and better scaling code. The results presented here show that this is indeed possible. However, it is also clear that some types of problems are much more suited to halo streaming than others.

Conditions that could encourage the use of halo streaming over halo exchange:

- **Small Local Work Buffer** - This was clearly seen in Figures 4.2 and 4.3 (visualisations of Vampir traces). When each process only has a small number of calculations to perform per iteration, halo streaming outperforms halo exchange by a large margin. This is because the streaming algorithm is much more efficient at overlapping the large amount of communication with the small amount of calculation. This is done both through reducing the total amount of communication, and removing restrictions on when messages need to complete.

- **Large Numbers of Processes** - As was shown in many of the results, halo streaming showed the most improvement over halo exchange when a very large number of processes was used. This is because increasing the number of processes increases the chance of delays. This affects halo streaming much less than it does halo exchange due to the lack of synchronisation.

- **Limited or Costly Resources** - One important result was the discovery that halo streaming needs only half as many processes as halo exchange to fit all of its data into cache (See Figure 4.11). This means that if resources are limited, halo streaming can be used to minimise the number of processors required, while still achieving good performance.
Conditions that could discourage the use of halo streaming over halo exchange:

- **Large Local Work Buffer** - As can be seen in Figure 4.4, halo streaming can still outperform halo exchange when a larger Work Buffer is used. However, the difference in performance is minimal, and is even likely to favour halo exchange in some cases. Because of this, it is most likely to be unnecessary to favour halo exchange in some cases. Halo exchange is much more simple to implement, making it the better option in this case.

- **Unlimited Resources** - This is not often the case for large-scale projects, but in the event of resources being (effectively) unlimited, it may be more sensible to achieve improved performance through the dedication of those resources, rather than taking the time to implement a halo streaming model.

- **Small Numbers of Iterations** - As was discussed in Section 4.2.3, if the number of iterations is small compared to the size of the Work Buffer, a limit is placed on how far a process can get ahead of its neighbours. Since part of the goal of halo streaming is to remove the dependency on other processes, it is clear that a problem that does not allow it to do this is not suited to the algorithm.

- **Short Problems** - Halo streaming takes considerably more effort to implement than halo exchange. The number of lines of code required is much greater, and the code itself is more complex. Because of this, halo streaming should only be used if it will save a significant amount of time in the long run. For small problems, the time spent implementing the model is likely to be much greater than the time saved by increasing communication efficiency.

- **Complex Problems** - Halo streaming cannot be adapted to different problem types as easily as halo exchange. This is largely because the rules that define halo exchange are much simpler. Domain decomposition problems can appear in almost infinitely many forms, with different kinds of stopping criteria, update rules, stencils, topologies etc. Each of these would require the halo streaming code to be modified in some (likely non-trivial) way. This is discussed further in Section 6.2.

Another aspect of halo streaming is the ability to tune its performance. The sizes of the Send and Receive Buffers, as well as the messages can be adjusted to give optimal performance. This is influenced by the nature of the problem, but also by hardware details such as cache size and communication bandwidth. This can be an advantage over halo exchange, which is much more restricted in the ways its parameters can be altered. However, it is also worth noting that a downside to this flexibility is a higher responsibility to choose the correct parameters. While choosing the correct parameters may give optimal performance, choosing incorrectly may give terrible performance. This was seen in Figure 4.12, which demonstrated the effect of message size on the execution time.

In summary, a halo streaming code was developed as an alternative to halo exchange. This code focused on using data streams and taking advantage of data dependencies in order to improve communication between processes. The code was tested and compared with halo exchange in
a variety of different situations on the Morar and ARCHER machines. The results have shown that halo streaming has the potential to be a huge improvement on halo exchange in many cases. The results consistently show improved performance, ranging from very slight benefits to an order of magnitude faster. However, it is also clear that halo streaming functions best in problems that fit certain criteria. These problems are characterised by having a large amount of communication relative to calculation, as well as being long enough allow for a significant amount of time to be saved.
Chapter 6

Future Work

6.1 Multidimensional Cases

This project focused entirely on one-dimensional domain decomposition problems. There is no reason why the algorithm could not be extended to an arbitrary number of dimensions, but there are a number of difficulties that prevent it from being a trivial task.

The first is the problem of diagonally adjacent neighbours. In the one-dimensional case, each process has at most two neighbours. Additionally, but the size of the boundary between each pair of processes are identical. However, this is no longer the case when the model is extended to multiple dimensions. Take, for example, a two-dimensional problem with a 9-point stencil (one of the simplest common multidimensional cases): To update cells that are on the domain boundary, data is needed from one of the horizontal or vertical neighbours. This data can be transferred in the same manner as it would be in the one-dimensional case, with the only difference being the size of the halo region. However, the corner data cells also require data from the diagonally adjacent processes. This causes problems even in the comparatively simple case of halo exchange. In order to pass this corner data to the correct destination, each process is forced to send four extra messages. This can double the communication time, while only contributing to the updating of a tiny fraction of the data. Halo exchange codes often circumvent this by packing their corner data into the horizontal communications. Having receiving this information, the corner data can then be repackaged into the vertical communications, which will arrive at the correct process.

This process can often be complicated to program into a halo exchange code, but it is even more complex to integrate it into the streaming model. One reason for this is that the method of performing the horizontal communications before the vertical forces a form of synchronisation on the processes. Processes cannot begin to send their vertical communications until the horizontal communications have arrived. Arranging the communications in this manner would defeat the purpose of using halo streaming.

A further difficulty concerns the determination of the order of calculations. In the one-dimensional
case each process only has one Send and Receive Buffer. It is easy to monitor the content of these and process data such that neither buffer ever becomes full. However, in the multidimensional case, there are exponentially more neighbours, and therefore buffers. Determining the order in which these buffers are drained may not be trivial. Choosing one buffer to drain at a time is the simplest option, but this may not give optimal performance. On the other hand, choosing the buffer that is the closest to being full may perform better, but it makes the task of keeping track of how many updates that have been performed on each cell much more complex.

6.2 Extension of Functionality

One difficulty with the halo streaming model is that associated with adapting to small changes in the nature of the problem. Because of this, more work will need to be done in order to enable common problem types to be solved using halo streaming. For example, altering a halo exchange code to reflect non-periodic boundary conditions is an extremely simple task. However, this is not the case with halo streaming. Since the domains of each process are constantly changing, each process must constantly keep track of the location of the boundary to ensure that none of the calculations use data from the opposite side.

Another common feature that could be added to the code is the ability to terminate the program based on the value of some global variable. This is easy to do with halo exchange, as the communication period can be used to perform a global reduction. For the streaming model, no iteration except for the last one is ever stored entirely in memory simultaneously. Therefore, it is not easy to track global properties of the system. In order to do this, the global reduction iterations must be identified ahead of time. Each process can store data from this iteration into a separate buffer that will not be overwritten. When this entire iteration is complete, the global reduction can be performed.

Similarly, the code does not adapt easily to include a stencil that includes a static background array. Using a halo exchange model, this array would be divided among processes in the same way as the Work Buffer, and used in each update. This cannot be done using halo streaming as the domain of each process changes over the course of the program. In order to allow the model to solve this kind of problem, one of two options must be chosen. First, the entire background array could be stored on every process. This would be simple to implement, with the only addition to the code being some minor bookkeeping to keep track of which section of the background array should be used. However, this would obviously be extremely expensive in memory, and would not be feasible for many problems. The second option is to transfer the necessary sections of the background array along with the halo data. This would increase the time taken to send each message, but likely not by a significant margin. The large amount of bookkeeping involved, as well as associated issues with cache use, causes significantly difficulties which would need to be addressed.
6.3 Circular Buffer

For this project, the algorithm was implemented in such a way that three buffers were used: one for performing calculations, one for storing data to be sent, and one for storing received data. However, it may be possible to combine these into a single circular buffer. The goal of this change would be to improve the flexibility of the code, allowing it to adapt to the availability of information more easily.

This buffer would work as follows: at the outset the buffer is to be partitioned in the same ratio that the current buffers have. The first section of the buffer is reserved for the send data, the middle section contains the initial data, and the final section is reserved for incoming data. The process uses pointers to store information about the location of the boundaries. When the code reaches a point where it begins to process the first piece of received data, the boundary between the Receive and Work sections is shifted to the other side of the received data. This essentially makes the received data part of the work section. The received information is then overwritten by the data it is used to update. When data is ready to be sent the boundary between the Send and Work regions is shifted to the other side of the data, making it part of the Send section. The result of this shifting is that the work section gradually consumes the receive section, with the send section gradually consuming the work section. When the Receive Buffer has been filled it will begin to use memory that was formerly part of the Send Buffer.

One advantage of this method is the flexibility of the structure. The boundaries can all move at different rates, so the process can prioritise the movement of boundaries that are too close to each other, or the draining of buffers that are too full. Additionally, this method has the potential to make more efficient use of memory, as the buffers that require more space will naturally consume parts of the buffers that do not need it.
Bibliography

[1] Archer hardware details. URL
   www.archer.ac.uk/about-archer/hardware/.


   www.epcc.ed.ac.uk/facilities/other-facilities/morar.


