Exploring XcalableMP

Shun Liang

August 24, 2012

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2012
Abstract

This project has implemented synthetic and application benchmarks for a novel parallel programming model called XcalableMP, which aims to provide convenient means for data parallel and PGAS programming. The results of the benchmarking running on the HECToR supercomputer have shown many features of XMP perform significantly worse than the equivalent features on HECToR. However, by doing the investigation, it is believed that several factors might have interfered the evaluation of the XMP code’s performance.
# Contents

## List of Figures

- Introduction i

## 1 Introduction

## 2 Background

- **XcalableMP**
  - Execution and Data Model 4
  - Features 4
- The XcalableMP Compiler 9
- Comparing XcalableMP with Other Programming Models
  - Message Passing Interface 10
  - High Performance Fortran 10
  - Coarray Fortran and Unified Parallel C 10
- Previous Benchmarking Work 11
- The HECToR Supercomputer 13

## 3 Microbenchmarking

- Method and Design of Microbenchmarks 14
  - Ping-Pong Microbenchmarking 14
  - Loop and Halo Swapping 16
  - Barrier Synchronisation 19
  - Broadcast 20
  - Reduction 21
- Results 22
  - Ping-Pong Microbenchmarking 22
  - Loop and Halo Swapping 22
  - Barrier Synchronisation 23
  - Broadcast 24
  - Reduction 25

## 4 Application Benchmarking

- Image Reconstruction 27
  - Algorithm 27
  - Implementations 28
  - Halo Swapping and Image Reconstruction 32
4.1.4 Results ................................................................. 34

5 Conclusion and Future Work ........................................ 37

6 Bibliography ................................................................. 39
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The code example (top) and the illustration (bottom) of the \textit{shadow} and \textit{reflect} directive</td>
<td>6</td>
</tr>
<tr>
<td>3.1</td>
<td>Results of the ping-pong microbenchmark set</td>
<td>22</td>
</tr>
<tr>
<td>3.2</td>
<td>Results of the parallel loop microbenchmark set</td>
<td>23</td>
</tr>
<tr>
<td>3.3</td>
<td>Results of the halo swapping microbenchmark set</td>
<td>23</td>
</tr>
<tr>
<td>3.4</td>
<td>Microbenchmarking Results for barrier synchronisation in XMP and MPI</td>
<td>24</td>
</tr>
<tr>
<td>3.5</td>
<td>Microbenchmarking Results for broadcast of a scalar variable in XMP and MPI</td>
<td>25</td>
</tr>
<tr>
<td>3.6</td>
<td>Microbenchmarkings result for broadcast of array in XMP and MPI</td>
<td>25</td>
</tr>
<tr>
<td>3.7</td>
<td>Microbenchmarking Results for reduction in XMP and MPI</td>
<td>26</td>
</tr>
<tr>
<td>4.1</td>
<td>An example of input image (left) and output image (right) of the reconstruction algorithm after 10000 iterations</td>
<td>28</td>
</tr>
<tr>
<td>4.2</td>
<td>Equation of the image reconstruction for the pixel $i,j$ in the image</td>
<td>28</td>
</tr>
<tr>
<td>4.3</td>
<td>Runtime result of image data scattering in the XMP and the MPI image</td>
<td>34</td>
</tr>
<tr>
<td>4.4</td>
<td>Runtime result of image data gathering in the XMP and the MPI image</td>
<td>35</td>
</tr>
<tr>
<td>4.5</td>
<td>Runtime result of halo swapping in the XMP and the MPI image processing programs</td>
<td>36</td>
</tr>
<tr>
<td>4.6</td>
<td>Runtime result of image reconstruction in the XMP and the MPI image</td>
<td>36</td>
</tr>
</tbody>
</table>
## Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Code example of data mapping and parallel loop in XMP</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Example code fragment of the <code>gmove</code> directive</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>Code example of coarray in XMP</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>Code fragment of the XMP ping-pong microbenchmark</td>
<td>15</td>
</tr>
<tr>
<td>3.2</td>
<td>Code fragment of the MPI ping-pong microbenchmark</td>
<td>15</td>
</tr>
<tr>
<td>3.3</td>
<td>Code fragment of the XMP parallel loop microbenchmark</td>
<td>16</td>
</tr>
<tr>
<td>3.4</td>
<td>Code fragment of equivalent implementation of 3.3 in MPI</td>
<td>17</td>
</tr>
<tr>
<td>3.5</td>
<td>Code fragment of the XMP halo swapping benchmark</td>
<td>17</td>
</tr>
<tr>
<td>3.6</td>
<td>Code fragment of the MPI implementation of the <code>reflect</code> directive</td>
<td>18</td>
</tr>
<tr>
<td>3.7</td>
<td>Code fragment of the XMP barrier synchronisation microbenchmark</td>
<td>19</td>
</tr>
<tr>
<td>3.8</td>
<td>Code fragment of the MPI barrier synchronisation microbenchmark</td>
<td>20</td>
</tr>
<tr>
<td>3.9</td>
<td>Code fragment of the XMP broadcast microbenchmark</td>
<td>20</td>
</tr>
<tr>
<td>3.10</td>
<td>Code fragment of the MPI broadcast microbenchmark</td>
<td>20</td>
</tr>
<tr>
<td>3.11</td>
<td>Code fragment of the XMP reduction microbenchmark</td>
<td>21</td>
</tr>
<tr>
<td>3.12</td>
<td>Code fragment of the MPI reduction microbenchmark</td>
<td>21</td>
</tr>
<tr>
<td>4.1</td>
<td>Code fragment of global arrays declaration for image data decomposition</td>
<td>29</td>
</tr>
<tr>
<td>4.2</td>
<td>Code fragment of image scattering in the XMP implementation</td>
<td>29</td>
</tr>
<tr>
<td>4.3</td>
<td>Code fragment of image gathering in the XMP implementation</td>
<td>30</td>
</tr>
<tr>
<td>4.4</td>
<td>Code fragment of image scattering in the MPI implementation</td>
<td>30</td>
</tr>
<tr>
<td>4.5</td>
<td>Code fragment of image gathering in the MPI implementation</td>
<td>31</td>
</tr>
<tr>
<td>4.6</td>
<td>Code fragment of halo swapping and image processing in the XMP implementation</td>
<td>32</td>
</tr>
<tr>
<td>4.7</td>
<td>Code fragment of halo swapping in the MPI implementation</td>
<td>32</td>
</tr>
</tbody>
</table>
Acknowledgements

I am grateful to Mr. Adrian Jackson at EPCC for his supervision and advices during this project. I would also thank Dr. Nakao Masahiro at the University of Tsukuba for providing information and helps to this project through the XMP mailing list.
Chapter 1

Introduction

Supercomputers nowadays are mostly deployed with distributed-memory architectures. Message Passing Interface (MPI) has been the dominating parallel programming model on distributed-memory architectures for two decades. [1] p. 39] MPI has been widely used by the scientific community and also the industry. The implementations of MPI provided by major vendors are also considered very mature after they have been used and evolving for years. However, since programmers have to explicitly manage both sending and receiving of communication in the code, MPI has the limitations which have introduced some difficulties for parallel programming and made bugs easier to be occur. Also, for the same reason, parallelising existing software usually involves reconstruction of the whole programs and demands more efforts from the MPI programmers. OpenMP is considered more convenient for parallelsing existing code, as the parallelisation is done by adding compiler directives to the code and does not involve reconstruction of the program. However this programming model only targets shared-memory architectures.

The supercomputing community has proposed several new programming models to make parallel programming on supercomputers more convenient. The concept of data parallelism has been introduced to represent parallel loop processing in a more compact format than the one in MPI. High Performance Fortran (HPF) is a data parallel extension to Fortran standardised at 1993 and it still has a considerable amount of users. [2] Partitioned Global Address Space (PGAS) is another parallel programming model which is proposed to implement flexible remote data access in parallel programs. The two major languages implementing PGAS are Coarray Fortran (CAF) and Unified Parallel C (UPC).

At 2010, a parallel programming extension to both C and Fortran called XcalableMP (XMP) was proposed. XMP targets distributed-memory computer architectures. By designing parallel programming features mainly as compiler directives, this language extension aims to make writing and optimising parallel code easier than the ways which
the existing solutions have provided. The design of XMP has incorporated both data parallelism and the PGAS programming model. Currently the implementations of the XMP compiler and the XMP programming library are premature and contributions are demanded for both development and benchmarking of XMP.

This project aims to investigate the performance of the XMP programming model by implementing benchmarks in XMP and comparing their performance with the MPI equivalent benchmarks. This dissertation presents the methods and achievements of the investigation completed by the project. Following this chapter, the background information of the XMP features, the comparisons of XMP and other programming models and previous researches is provided in Chapter 2. Chapter 3 presents the designs and the runtime results of the synthetic benchmarks (microbenchmarks) of a set of features in XMP, and compares the performance of the XMP microbenchmarks with the equivalent implementations in MPI. Chapter 4 describes the implementations of an application benchmark implemented in both XMP and MPI and compares the runtime results of the two implementations. Finally, the conclusion and possible future work of this project are provided in Chapter 5.
Chapter 2

Background

2.1 XcalableMP

XcalableMP is a directive-based parallel programming model designed and developed by a group of researchers in the University of Tsukuba and other institutions of the academia and the industry in Japan. The programming model is fully described in its version 1.0 specification. When this dissertation is being written, the newest release of the XcalabeMP compiler is version 0.5.4, which is still in alpha release and has not yet fully supported the features in the specification. A modified version of 0.5.4, which is called 0.5.4-1 and has some known bugs fixed, has been provided from the XMP developers to this project. The documentations, compiler releases and other information are available at the XcalableMP web site.

The design of XcalableMP incorporates global-view programming features to support data parallelism and work mapping, and local-view programming features to support PGAS programming. Global-view programming is designed to be done by writing compiler directives. In XMP, different sections of a global array can be mapped to different processors, and data movements across processors can be easily done. Loop parallelism, collective operations and barrier synchronisation are included in the global-view programming features. The local-view programming mode of XMP, which aims to support PGAS programming, is designed to be similar to the Coarray Fortran programming model.

This chapter describes the features of XMP in details. It also compares the design of XMP and other parallel programming models including MPI, HPF, CAF and UPC. The status and limitation of the current XMP compiler release is provided. Previous XMP benchmarking works is also presented. Finally, HECToR, the supercomputer used for benchmarking by this project, is described with technical details.
2.1.1 Execution and Data Model

Recall that the target platforms of XcalableMP are distributed-memory architectures. The execution model of XcalableMP is Single Program Multiple Data (SPMD), which implies that there are multiple processes running the same program in parallel on the nodes. Processes are running without synchronisation and communication until the executions encounter a directive which handles the synchronisation and communication.

The XMP specification defines a node as identical to an MPI process, which is a replication of the running instance of the parallel program, although it is possible to allocate multiple XMP nodes on one physical node of the supercomputers. An execution node set refers to a set of nodes running in a procedure or encountering a directive of a context. When an XMP program starts at the main procedure (main function in C), the current execution node set is the set of all nodes running in the program. [3, p. 4]

In global-view programming, data are classified as global or local in XMP. Global data includes arrays and chunks of allocated memory which are aligned to a set of nodes. By aligning the global data to the node set, the global data will have a fraction stored by each node’s memory. A node can only access to the fraction of global data which is aligned to it. Scalar variables, and arrays and chunks of memory which are not global are classified as local data. Since the programming model of XMP is SPMD, the definitions of local data are replicated on all nodes in a program.

Data parallelism is achievable with the global-view XMP programming model. For example, when an execution node set encounters a loop directive followed by a loop statement, the loop execution will be parallelised as each of node will complete the loop tasks which involve access to its fraction of the global data. Directives which handle synchronisations and communications have also made data parallel programming more convenient.

In local-view programming of XMP, a global array is composed by a set of local arrays or scalar variables residing on different nodes in the node set in a bottom-up manner. Instead of using directives to achieve data parallelism, local-view programming allows any node to access any section of the global data, no matter which node the section is allocated on. This is similar to the programming approaches in CAF and UPC.

2.1.2 Features

Recall that several features have been designed in the specification of XcalableMP to support data parallel and also coarray programming. These features have been designed
in three forms: compiler directive, library procedures/functions and extensions of base languages’ syntax. The XMP directives have been classified as declarative directives and execution directives. A declarative directive is one which handles global data declaration and alignment, while an execution directive is one which performs an action in runtime. There is a special kind of execution directives, which needs to appear immediately before a statement in the program to fulfil the semantics of the directives. These directives are classified as constructs in XMP. A XMP library has also been designed to provide procedures/functions to retrieve the information of the node and also allocate global data dynamically. Finally, the syntax of the base languages of XMP, especially C, has been extended to support array section notation and coarray programming.

Data Mapping

In order to achieve data parallelism, XMP provides means to map global data to a node set with several XMP declarative directives. The first one is the node directive. The node directive declares a node set which is used for data and work mapping. The template directive declares a template, which is a dummy array used for global data mappings.

A template can be distributed onto a node set with different distribution styles, and this is done by the distribute directives. There are several distribution formats of the distribute directives. The block format distributes the global data as contiguous memory blocks onto the node set. The cyclic format distributes the global data in a round-robin style onto the node set. When an integer \( n \) is appended to the cyclic directive, the global data is distributed a round-robin style with \( n \) contiguous elements as a unit of distribution. The third format, gblock, takes an array \( m \) as argument and distributes the global data in the manner that allocates \( m[i] \) number of contiguous elements onto the \( i_{th} \) node in the node set.

In XMP, an array is declared as global by being aligned to a template. The global array will be distributed onto the node set in the same manner as the aligned template is distributed. The align directive handles the global array declaration. This directive needs to appear immediately after the array declaration statement in the base language to fulfil the global declaration. Global arrays defined with the align directive are able to be processed in parallel with with the loop directive, which is described in the work mapping sub-subsection. An example of data mapping with align is shown in figure 2.1.

As a node can only have access to its local section of the global array, XMP has provided means for the nodes to store copies of the boundary elements of each node’s neighbouring nodes’ local sections of the global array. The storage of the neighbouring
boundary elements is handled by the shadow directives. These extra storages are conventionally called “halos”. A shadow directive declares halos for each local section of the global array, and the size of halos can be specified by the programmer when writing the shadow directive. The halo defined by the shadow directive can be exchanged among the nodes by the reflect directive. This process is conventionally called “halo swapping”. An example of the shadow and reflect directives and the example code’s effect is shown in figure 2.1. In this example, the array `array` which has 16 elements in total has been aligned to the node set `p` which has 4 nodes, hence each local array section has 4 elements to store the array elements. Since the shadow directive has declared two more extra elements of each section, one before the first array element, and one after the last array element, to store the boundary element of the two neighbouring sections, each local section of `array` stores 6 integers. When the reflect directive is encountered by the nodes, each node sends its first array element to the right halo of the node’s left neighbouring node, and sends its last array element to the left halo of the node’s right neighbouring, at illustrated in the figure.

Work Mapping

Work mapping is supported by a set of XMP features. The loop construct is one of the most important work mapping directive as it maps the loop iteration tasks to the node set in the manner which the global array is aligned in. An example showing work mapping with the loop directive is shown in listing 2.1. The task directive is another work mapping directive. This can be used more flexibly than the loop construct as the statements following this construct are executed on any node in the node set specified in the code. A set of different task constructs can be wrapped by the tasks construct in the code and then runs concurrently.

Listing 2.1: Code example of data mapping and parallel loop in XMP
Communication and Synchronisation

A set of XMP directives are provided to implement communication between and synchronisation among nodes. Although there is no means for explicit communication such as `MPI_Send` and `MPI_Recv` in XMP, data can still be moved across nodes with several directives. The `gmove` construct allows assignment between sections of global or local arrays, with the extension of array section indexing to the C syntax. When executing the assignment statement following the `gmove` directive, an implicit synchronisation is performed at the end of the assignment, unless the `async` keyword is appended to the `gmove` construct. An example code fragment of `gmove` is shown at listing 2.2

```
/* Declare node set "p" and template "t" */
#pragma xmp nodes p(*)
#pragma xmp template t(0:N−1)
#pragma xmp distribute t(block) onto p
/* Declare global array "array" */
int array[N];
#pragma xmp align array[i] with t(i)
int i;
/* Mapping loop tasks to the node set */
#pragma xmp loop(i) on t(i)
for(i = 0; i < N; i++)
{
    array[i] = i;
}
```

Listing 2.2: Example code fragment of the `gmove` directive

```
#pragma xmp nodes p(4)
/* N should be divisible by 4 */
#pragma xmp template t(0:N−1)
int left_array[N], right_array[N];
#pragma xmp align left_array[i] with t(i)
#pragma xmp align right_array[i] with t(i)
#pragma xmp gmove
left_array[0:N/4−1] = right_array[3*N/4:N−1];
```

Broadcast and reduction have also provided in XMP. Broadcast is implemented with the `bcast` directive, which can be used to broadcast scalar variables and arrays from one node to other nodes in the node set. The `reduce` directive performs a reduction operation
on a set of global arrays, and the reduction result is replicated on all nodes in the current executing node set after the operation. This is equivalent with \texttt{MPI\_Allreduce}. The reduction operation can be summation($\Sigma$), product($\Pi$), bitwise operator, logical operator and finding minimum and maximum. The \texttt{reduce} directive can also be appended to a \textit{loop} directive, and in this case the reduction is performed at the end of the parallel loop.

Barrier synchronisation is also provided in XMP. The \texttt{barrier} directive performs a barrier synchronisation for all nodes encounters the directive. This is equivalent to \texttt{MPI\_Barrier}.

\textbf{Local-view Programming}

The local-view programming mode in the XMP specification aims to PGAS programming. It follows the design of Coarray Fortran. XMP programmers can declare an array as a coarray allocated on nodes in the node set with the \texttt{coarray} directive. A coarray is an array which is allocated globally in a bottom-up manner and allows any node in the node set to access any element local or remotely. A node possessing a section of a coarray is called an image in the coarray programming context. An example code fragment borrowed from the one in the XMP specification is provided in listing 2.3. In the example program, two coarrays which have 10 elements at the local section on each image have been declared. At the last line of the code, the elements on the local section of B on image 10 have been copied to the elements on all local sections of the coarray A. Some XMP directives have also been defined in the XMP specification to allow memory synchronisation of coarrays among images and point to point synchronisation in local-view programming. However, the entire local-view programming features are not yet supported by the current release of the XMP compiler.

\begin{center}
\textbf{Listing 2.3: Code example of coarray in XMP}
\end{center}

\begin{verbatim}
#pragma nodes p(16)
int A[10], B[10];
#pragma xmp coarray A, B : [*]
A[:] = B[:]:[10];
\end{verbatim}

\textbf{XMP Library Functions and Base Language Extensions}

A set of auxiliary library functions (subroutines in Fortran) and operators has been provided in the XMP specification. Some of these functions and operators provide support to data mapping, some provide support to node identification, and some provide support to timing.
The `align` construct makes a statically declared array allocated globally on the node set. The XMP specification also defines the `xmp_malloc` to support global dynamic memory allocation. However, this is not yet supported by the current release of the XMP compiler (0.5.4-1).

A set of functions are provided in the XMP library to retrieve the information of the entire node set, the current executing node set and the node itself. The `xmp_all_num_nodes` function returns the number of nodes in the entire node set. The `xmp_all_node_num` give the index of the node which invokes this function in the entire node set. The `xmp_num_nodes` and `xmp_node_num` return similar information but of the current executing node set. These functions are similar to the `MPI_Comm_size` and `MPI_Comm_rank` functions in MPI.

Two functions have been defined in the XMP specification to support timing in the code. The `xmp_wtime` function returns the elapsed time in seconds since “some moment in the past” in the program. The `xmp_wtick` returns the resolution of `xmp_wtime`. These two functions provide are equivalent to the `MPI_Wtime` and `MPI_Wtick` in MPI. However, the `xmp_wtick` is not ye implemented in the currently release of the XMP library implementation (0.5.4-1).

### 2.2 The XcalableMP Compiler

Compiling XMP program involves two stages. At the first stage, the XMP compiler reads the XMP source code, and translates the XMP source code to the code in the base language (C or Fortran). Directives in the source code are translated as XMP library function calls. The communication functions in the XMP library are implemented as one-sided communications in MPI, which include `MPI_Get`, `MPI_Put` and `MPI_Accumulate`. The XMP compiler also calculate the the array index bound and the memory addresses on each node when translating XMP directives which achieve data parallelism. The second stage of compiling passes the translated base-language code to the native compiler and compiles it to object code. Currently the XMP compiler can only work with GCC as native compiler.

There are several limitations of the current release of the XMP compiler (0.5.4-1). First, despite that the XMP specifies extensions to both C and Fortran, the compiler currently only supports C. Second, the compiler has not yet supported the `gblock` distribution for distributing templates onto node sets. Third, the `tasks` directive which lets different tasks to run in parallel is not yet supported. Finally, the global memory allocation is not yet implemented.
2.3 Comparing XcalableMP with Other Programming Models

Comparisons of the design of XcalableMP and other programming models have been provided in this section. The comparisons focus on two aspects of these programming models: ease and flexibility of writing parallel programs.

2.3.1 Message Passing Interface

The previous section has shown that many XMP features are equivalent to the features in MPI. However, due to the inherent difference of the designs of these two programming models, the ways of programming in XMP and in MPI fundamentally differ to each other. In contrast to the implicit way of communications in XMP, communications in MPI have to be performed with explicitly specified senders, receivers and sizes of message. Also, MPI provides no built-in support for data parallelism, while programmers can write data parallel code without too much overhead with the XMP features such as the align and the loop directive. However, writing non-data parallel program in XMP might be difficult since its communication features are not as flexible as the ones in MPI.

2.3.2 High Performance Fortran

The design of XMP has been significantly influenced by High Performance Fortran. Both XMP and HPF are directive-based parallel programming models targeting distributed-memory architectures. The syntax of data parallelism is similar in these two programming models. For example, they both have the template representation to be distributed onto the nodes and can be aligned with global arrays. Also, they both have directive-based support for data parallelism. The loop directive in XMP is equivalent to the FORALL directive in HPF. However, as stated by the developers of the XMP compiler, inter-node communication is not automatically performed by the XMP compiler, while it is done in the opposite way in HPF. This implies writing XMP code involves using communication directives explicitly. This needs more attentions from the programmers, but allows them to optimise the inter-node communication. Lastly, as implied by its name, HPF is an extension to Fortran only, while XMP aims to support both Fortran and C, although the current release of compiler only supports C.

2.3.3 Coarray Fortran and Unified Parallel C

XMP has been compared to two PGAS languages: Coarray Fortran and Unified Parallel C. The specification of XMP has incorporated coarray as a part of local-view programming features, and the notations of XMP coarray programming is very similar to
the notations in CAF. Also, the specification of XMP has provided memory synchronisation features, which are compatible with the ones in CAF. Also, the barrier synchronisation feature in XMP are equivalent to the ones in CAF, as barrier is equivalent to syscall and syncimages. It is expected that programs written in CAF will be easily translated to local-view code in XMP Fortran when local-view programming and Fortran as the base-language is supported by the XMP compiler. However, as stated in the previous discussion, CAF only extends the Fortran programming language, while XMP aims to support both Fortran and C.

UPC is another programming model which follows the PGAS design to implement data parallelism in a bottom-up manner. UPC has provided the shared keyword to declare global arrays in the code. However, different to the global arrays which are declared with align in XMP, local sections of shared global array in UPC can be accessed by any node (which is called “thread” in UPC) in the node set remotely, while gmove or reflect has to be used while access data on other nodes in XMP. A shared array in UPC is distributed in the cyclic manner in default, and the length of the unit of cyclic distribution can be specified in the code. However, it is considered less flexible than the array distribution in XMP, which has also provided block distribution and generalised block distribution (gblock), despite the fact that gblock is still not supported by the current release of the XMP compiler. UPC has also supported data parallel programming with the upc forall statement, which is similar to the loop directive in XMP. Similar to the communication and synchronisation directives in XMP, UPC has provided collective operators and barrier synchronisation.

In conclusion, remote data access is considered more flexible in PGAS languages rather than in global-view programming of XMP. However, some features of XMP is considered more convenient than the explicit remote data access features in PGAS languages. For example, for applications involving halo swapping, programmers need to specify the indexing of the partitioned global data, while in XMP this can be done with the one-line reflect directive.

2.4 Previous Benchmarking Work

The developers of the XMP programming model have completed some investigations of the performance and productivity of XMP. One of their publications has provided results of the XMP implementation of the STREAM benchmark, the Linpack benchmark and the FFT benchmark, on the T2K Tsukuba System. In the benchmark program, the STREAM benchmark calculated \( a = b + \text{scalar} \times c \), while \( a, b \) and \( c \) were vectors, and \( \text{scalar} \) was a scalar variable. The Linpack benchmark measured the number of floating point operations per second of the program solving a large-size dense systems of linear equations. The dgefa and dgesl Linpack routines had been selected to parallelise
in XMP. The $gmove$ directive was significantly involved to process the pivot buffer in the benchmark. The XMP FFT benchmark perform Fast Fourier Transform took the transpose of matrix as a key step, and this was implemented with the $gmove$ directive in XMP. The runtime results of these benchmarks have shown the performance of the STREAM benchmark speeded-up linearly when the number of nodes increased from 2 to 32; The performance of the XMP Linpack benchmark was not satisfying as the FLOPS grew sub-linearly when the number of nodes increased; The performance of XMP FFT grew in a logarithmic manner when the number of nodes increased from 2 to 32. [6]

Another work has been done to compare the performance of the XMP an the MPI implementation of the NAS parallel benchmark, which was originally implemented in MPI, on the T2K Tsukuba System. Three kernels in NAS has been implemented in XMP: Embarrassingly Parallel (EP), Integer Sort (IS) and Conjugate Gradient (CG). The EP benchmark calculated pseudo-random numbers in parallel and did not involve any communication between nodes. The IS benchmark sorts large integer arrays based on the bucket sort algorithm. It had two versions of XMP implementation: IS with a histogram and IS without a histogram. Despite the fact that the README file has stated local-view programming has not yet been supported by the currently release, the IS with a histogram involved coarrays in the program, and the IS without a histogram was implemented with pure global-view programming. The IS with a histogram involved all-to-all communication among the nodes, and this was implemented with the $gmove$ directive in the XMP and $MPI_{Alltoall}$ in the original version. The CG benchmark measured the runtime of the programs computing the smallest eigenvalue of a sparse symmetric positive finite matrix. It has been implemented with 1-dimensional and 2-dimensional decomposition in XMP and has involved the $reflect$, $reduction$ and $gmove$ directives. The runtime results of the NAS benchmark have shown that the FLOPS of both the XMP and MPI versions of EP grew when the number of nodes increased from 1 to 16, however the FLOPS of the XMP version kept being only half of the FLOPS of the MPI version; The performance of the IS with a histogram was similar to the performance of the MPI version, but the IS without a histogram in XMP performed very poorly; The performance of the CG benchmark in XMP with 1-dimension and 2-dimension decomposition were not very different with the performance of the MPI version. [10]

The third work has been done to compare the performance of code written in XMP and in UPC. The comparison has been made based on the results of three benchmarks: global data access, Laplace solver and Conjugate Gradient. The first benchmark was implemented with the $align$ and the $loop$ directive in XMP, and global array declaration and $upc_{forall}$ in UPC, for global arrays aligned in different manners. The Laplace solver involved accessing the neighbouring elements for each element in the 2-dimensional array. This benchmark has been implemented with the $loop$ and the $reflect$ directives to process the elements in parallel and exchange the boundary elements, and
the equivalent program was implemented with `upc_memget` and `upc_forall` in the UPC version. And the Conjugate Gradient benchmark was implemented with remote data accessing, `upc_memcpy` and `upc_memget` in the UPC version. The runtime results have shown that the XMP global data accessing benchmark significantly outperformed the version of UPC; The performance of the Laplace solver was much better in XMP than in UPC; the performance of the XMP and the UPC versions of Conjugate Gradient were similar. [11]

2.5 The HECToR Supercomputer

The HECToR supercomputer is used for all benchmarking during this project. The supercomputer employs the Cray XE6 System, which has 2816 number of compute nodes in total. Each node has two 2.3GHz AMD Opteron Interlagos processors, and each processor has 16 cores. Each processor is associated with 16 GB memory. The processors of HECToR are interconnected in the 3D-torus manner, and every two compute node is connected by one Gemini router chip. The bandwidth for MPI point-to-point communication is at least 5 GB/s, and the latency between two nodes is from 1 to 1.5 µs. [12]

The software packages installed on HECToR which have been used by this project include XMP compiler (ver 0.5.4-1), GCC (ver 4.6.3, used as base language compiler by the XMP compiler), Java (ver 1.7, used for code translation) and MPICH2 (ver 5.4.5).
Chapter 3

Microbenchmarking

In order to study the low-level performance of the XcalableMP implementation, a set of microbenchmark has been implemented and has run on both NESS and HECToR. This chapter presents the description and the runtime results of the microbenchmark set.

3.1 Method and Design of Microbenchmarks

A subset of XMP features was selected to benchmark. The microbenchmarking for each feature was completed by repeatedly invoking the XMP feature for many times, and run the MPI version the microbenchmarking code which completes the same tasks with the same number of iterations and finally comparing the performance of the XMP and MPI code which handles the same tasks. The selected features include barrier synchronisation, broadcast, ping-pong, parallel loop, halo swapping and reduction. The timing of the microbenchmark set is done by invoking the xmp_wtime function in XMP code and MPI_Wtime in MPI code. By investigating the source code of the currently implementation of the XMP library, the xmp_wtime function invokes MPI_Wtime and returns its return value.

3.1.1 Ping-Pong Microbenchmarking

An XMP ping-pong program and an MPI ping-pong program have been written to benchmark the array assignment feature in XMP, which is implemented as the gmove directive. The XMP ping-pong benchmarking program gets the first and the last node to assign the value of the elements of the local section of global array from each other’s local section of the global array. An equivalent MPI benchmark has been implemented with MPI_Send and MPI_Recv and runs the same number of iterations as the XMP version does. Code fragments of of these programs have been provided in listing 3.1 and listing 3.2.
Listing 3.1: Code fragment of the XMP ping-pong microbenchmark

```c
/* send_array and recv_array are global arrays */
/* which has been aligned and initialised */
/* Calculate the ending index of the local array */
/* section of the first node */
first_end_idx = ARRAY_SIZE / xmp_all_num_nodes() - 1;
/* Calculate the starting index of the local array */
/* section of the last node */
last_start_idx = (xmp_all_num_nodes() - 1) *
   (ARRAY_SIZE / xmp_all_num_nodes());
total_t = 0;
for (i = 0; i < REPEAT; i++)
{
    start_t = xmp_wtime();
    /* Ping */
    #pragma xmp gmove in
    recv_array[ARRAY_START: first_end_idx] =
    send_array[last_start_idx: ARRAY_SIZE - 1];
    /* Pong */
    #pragma xmp gmove in
    recv_array[last_start_idx: ARRAY_SIZE - 1] =
    send_array[ARRAY_START: first_end_idx ];
    end_t = xmp_wtime();
    total_t += end_t - start_t;
}
```

Listing 3.2: Code fragment of the MPI ping-pong microbenchmark

```c
/* send_buf and recv_buf are size chuck_size buffers */
/* allocated on each rank */
/* proc is the total number of ranks */
total_t = 0;
for (i = 0; i < REPEAT; i++)
{
    start_t = MPI_Wtime();
    /* Ping */
    if (0 == rank)
        MPI_Send(send_buf, chuck_size, MPI_INT,
                 proc - 1, 0, comm)
    else if (proc - 1 == rank)
        MPI_Recv(recv_buf, chuck_size, MPI_INT,
                  0, 0, comm, &status);
    /* Pong */
    if (0 == rank)
        MPI_Recv(recv_buf, chuck_size, MPI_INT,
```
3.1.2 Loop and Halo Swapping

Recall that XMP provides the loop directive for data parallelism, and the shadow and reflect directives for swapping halo data among the decompositions in an XMP program. These directives have been used in the programs which benchmark performance of parallel looping and halo swapping. There are two microbenchmarking set for these XMP features, one benchmarks the overhead of an XMP parallel loop and compares it to the runtime of the equivalent implementation in MPI; one benchmarks the overhead of an XMP parallel loop with halo swapping and compares it to the runtime of the equivalent implementation using \textit{MPI\_Send} and \textit{MPI\_Recv}.

The code fragment of loop microbenchmark without halo swapping is shown in listing 3.3 and listing 3.4. This set of microbenchmarks data parallel processing by using the loop directive in XMP and decomposing the data onto all ranks in MPI. Since there is no synchronisation and communication in the MPI program, the overhead of the loop directive can be measured by the difference the runtime of the XMP and MPI program.

**Listing 3.3: Code fragment of the XMP parallel loop microbenchmark**

```c
#pragma xmp nodes p(*)
#pragma xmp template t(0:ARRAY_SIZE-1)
#pragma xmp distribute t(block) onto p
array[ARRAY_SIZE];
#pragma xmp align array[k] with t(k)
#pragma xmp align
for (i=0; i<REPEAT; i++)
{
    start_t = xmp_wtime();
#pragma xmp loop (j) on t(j)
    for (j = 0; j < ARRAY_SIZE; j++)
    {
        for (k = 0; k < delay_length; k++)
        {
            array[j] += 1;
        }
    }
}
```
The other set of microbenchmark has been based on the code above, but focuses on measuring the time spent on halo swapping instead of the overhead of parallel loop. In the XMP microbenchmarking program, the declaration of the halo is handled by the _shadow_ directive, and the swapping of halo data is implemented with the _reflect_ directive. Listing 3.5 provides the code fragment of the XMP halo swapping program. An MPI version of this program has been implemented and uses _MPI_Send_ and _MPI_Recv_ to swap halo data. Listing 3.6 shows the MPI halo swapping code fragment. The performance difference of using the _reflect_ directive and standard message passing in MPI will be measured by the difference of the runtime of these two microbenchmarking programs.

Listing 3.5: Code fragment of the XMP halo swapping benchmark

```
#pragma xmp nodes p(*)
int array_size = ARRAY_SIZE;
int halo_length = HALO_LENGTH;
#pragma xmp template t(0:ARRAY_SIZE-1)
#pragma xmp distribute t(block) onto p
int array[ARRAY_SIZE];
#pragma xmp align array[k] with t(k)
#pragma xmp shadow array[halo_length]
```
int buffer[ARRAY_SIZE];
#pragma xmp align buffer[k] with t(k)
#pragma xmp shadow buffer[halo_length]

/* Initialise array */
#pragma xmp loop (i) on t(i)
for (i=0; i<ARRAY_SIZE; i++)
{
    array[i] = i;
}
total_t = 0;
#pragma xmp loop (i) on t(i)
for (i=0; i<REPEAT; i++)
{
    start_t = xmp_wtime();

    /* Swap halo data */
    #pragma xmp reflect (array)
    
    #pragma xmp loop (j) on t(j)
    for (j = 0; j < ARRAY_SIZE; j++)
    {
        if (0 == j)
            buffer[j] = -1 + array[j + 1];
        else if (ARRAY_SIZE - 1 == j)
            buffer[j] = array[j - 1] + ARRAY_SIZE;
        else
            buffer[j] = array[j - 1] + array[j + 1];
    }
    #pragma xmp loop (j) on t(j)
    for (j = 0; j < ARRAY_SIZE; j++)
    {
        array[j] = buffer[j];
    }
    end_t = xmp_wtime();
    total_t = end_t - start_t;
}

Listing 3.6: Code fragment of the MPI implementation of the reflect directive in listing 3.5

if (1 == rank % 2)
{
    /* Send to left neighbour */
    if (0 < rank)
        MPI_Send(&array[1], halo_length,
                MPI_INT, rank - 1, 0, comm);
3.1.3 Barrier Synchronisation

The microbenchmarking set on barrier synchronisation has been implemented with the XMP barrier directive and the MPI_Barrier function. Listing 3.7 and listing 3.8 show the code fragments of the microbenchmarking programs for barrier synchronisation in XMP and MPI.

Listing 3.7: Code fragment of the XMP barrier synchronisation microbenchmark

```c
/* Send to right neighbour */
if (proc - 1 > rank)
    MPI_Send(&array[chunk_size], halo_length,
             MPI_INT, rank + 1, 0, comm);

/* Recv from right neighbour */
if (proc - 1 > rank)
    MPI_Recv(&array[chunk_size + 1], halo_length,
              MPI_INT, rank + 1, 0, comm, status);

/* Recv from left neighbour */
if (0 < rank)
    MPI_Recv(&array[0], halo_length,
              MPI_INT, rank - 1, 0, comm, status);

} else {
    /* Recv from right neighbour*/
    if (proc - 1 > rank)
        MPI_Recv(&array[chunk_size + 1], halo_length,
                  MPI_INT, rank + 1, 0, comm, status);

    /* Recv from left neighbour */
    if (0 < rank)
        MPI_Recv(&array[0], halo_length,
                  MPI_INT, rank - 1, 0, comm, status);

    /* Send to left neighbour*/
    if (0 < rank)
        MPI_Send(&array[1], halo_length,
                 MPI_INT, rank - 1, 0, comm);

    /* Send to right neighbour */
    if (proc - 1 > rank)
        MPI_Send(&array[chunk_size], halo_length,
                 MPI_INT, rank + 1, 0, comm);
}

3.1.3 Barrier Synchronisation

The microbenchmarking set on barrier synchronisation has been implemented with the XMP barrier directive and the MPI_Barrier function. Listing 3.7 and listing 3.8 show the code fragments of the microbenchmarking programs for barrier synchronisation in XMP and MPI.

Listing 3.7: Code fragment of the XMP barrier synchronisation microbenchmark

total_t = 0;
for (i=0; i<REPEAT; i++)

19
3.1.4 Broadcast

A set of microbenchmarking has been implemented for the XMP broadcast feature. One broadcasts a scalar variable from a node to the whole node set; one broadcasts a local array from a node to the whole node set. These microbenchmarking programs use the \texttt{bcast} directive in XMP and the \texttt{MPI\_Bcast} function in MPI. The microbenchmarking code fragments for broadcasting a scalar variable is shown in listing 3.9 and listing 3.10. Microbenchmarking programs for array broadcasting are similar to these pair of code fragments.

Listing 3.9: Code fragment of the XMP broadcast microbenchmark

```c
double bcast_var, bcast_ref;
bcast_ref = 0;
bcast_var = xmp_all_node_num();
total_t = xmp_wtime();
for (i=0; i<REPEAT; i++)
{
    start_t = xmp_wtime();
    /* Broadcast */
#pragma xmp bcast (bcast_var) from p(0) on p(:)
    end_t = xmp_wtime();
    total_t += end_t - start_t;
}
```

Listing 3.10: Code fragment of the MPI broadcast microbenchmark

```c
double bcast_var, bcast_ref;
bcast_ref = 0;
bcast_var = xmp_all_node_num();
total_t = MPI_Wtime();
for (i=0; i<REPEAT; i++)
{
    start_t = MPI_Wtime();
    MPI_Barrier(comm);
    end_t = MPI_Wtime();
    total_t += (end_t - start_t);
}
```
```c
double bcast_var, bcast_ref;
MPI_Comm_rank(comm, &rank);
bcast_ref = 0;
bcast_var = rank;
total_t = 0;
for (i = 0; i < REPEAT; i++)
{
    start_t = MPI_Wtime();
    MPI_Bcast(&bcast, 1, MPI_DOUBLE, 0, comm);
    end_t = MPI_Wtime();
    total_t += end_t - start_t;
}
```

### 3.1.5 Reduction

The microbenchmarking set for the reduction feature is implemented with the `reduce` directive in XMP and the `MPI_ALLreduce` function. The code fragments of the XMP and MPI microbenchmarking programs are shown in listing 3.11 and listing 3.12.

Listing 3.11: Code fragment of the XMP reduction microbenchmark
```
double scalar;
scalar = xmp_all_node_num() + 1;
total_t = 0;
for (i = 0; i < REPEAT; i++)
{
    start_t = xmp_wtime();
    /* Reduction */
    #pragma xmp reduction(+: scalar) on p(:)
    end_t = xmp_wtime();
    total_t += end_t - start_t;
}
```

Listing 3.12: Code fragment of the MPI reduction microbenchmark
```
scalar = xmp_all_node_num() + 1;
total_t = 0;
for (i = 0; i < REPEAT; i++)
{
    start_t = MPI_Wtime();
    MPI_Allreduce(&scalar_s, &scalar_r, 1,
                  MPI_DOUBLE, MPI_SUM, comm);
    end_t = MPI_Wtime();
    total_t += end_t - start_t;
}
3.2 Results

3.2.1 Ping-Pong Microbenchmarking

The runtime results of the 1000-iteration ping-pong microbenchmarks have shown a significant performance difference between the XMP *gmove* directive and the *MPI_Send* and *MPI_Recv* functions in MPI. When the ping-pong message size (number of integers) is 32768, the runtime of the XMP microbenchmark was about 2.67 seconds, and the runtime of the MPI one was less than 0.1 seconds. When the message size increased to 2097152, the runtime of the XMP implementation was about 29.06 seconds, while the runtime of the MPI implementation was about 2.96 seconds. Also, the runtime results have shown the runtime of the XMP ping-pong microbenchmark dropped when the ping-pong message size increased from 32768 to 65536. The results are plotted in figure 3.1.

![Figure 3.1: Results of the ping-pong microbenchmark set](image)

3.2.2 Loop and Halo Swapping

Results of the microbenchmarks for parallel loop and halo swapping have been shown in figure 3.2 and figure 3.3. The first figure shows the runtime of the 1000-iteration parallel loop microbenchmarks decreased in log of 2 manner when the number of processes increased. The performance difference between the XMP version and the MPI version of parallel loop has been shown to be negligible. The second figure shows the runtime results of the 10000-iteration halo swapping microbenchmarks. The runtime of both programs decreased in logarithmic manner when the number of processes increased. However, a significant performance difference of the halo swapping programs in XMP
and in MPI has been revealed in the results as the XMP version always consumed more
time than the MPI one. This has shown the overhead of the reflect directive was more
than the overhead of $MPI_{\_Send}$ and $MPI_{\_Recv}$ on HECToR.

![Figure 3.2: Results of the parallel loop microbenchmark set](image)

![Figure 3.3: Results of the halo swapping microbenchmark set](image)

### 3.2.3 Barrier Synchronisation

The results of the 100000-iteration barrier synchronisation microbenchmarks have
been shown in figure [3.4]. As the number of processes running on HECToR increased
from 32 to 2048, the time spent on barrier synchronisation increased linearly for both the XMP and the MPI microbenchmarks. However, the performance of the XMP barrier synchronisation has been shown to be significantly worse than the one of the MPI equivalent implementation.

### 3.2.4 Broadcast

The results of the 100000-iteration scalar variable broadcast microbenchmarks have been shown in figure 3.5 and the results of the 100000-iteration array broadcast microbenchmarks have been shown in figure 3.6. For the scalar variable broadcast, both of the runtime of the XMP and the MPI implementation grew in a logarithmic manner when the number of processes increased from 32 to 2048 on HECToR, and the runtime of the MPI version is slightly less than the runtime of the XMP version. However, the difference of runtime was much more significant between the XMP and the MPI implementations of the array broadcast microbenchmark as the runtime of the XMP version was about 50.36 seconds, while the one of the MPI version was 9.72 seconds, when running on 2048 processes on HECToR.
3.2.5 Reduction

The results of the 100000-iteration reduction microbenchmarks have been shown in figure 3.7. The performance of the XMP implementation of the microbenchmark is considered significantly worse than the performance of the MPI implementation as the runtime of the XMP one grew much faster than the runtime of the MPI one when the
number of processes increased. When the number of process was 2048, the runtime of the XMP version was about 46.60 seconds, while the runtime of the XMP version was about 3.92 seconds.

Figure 3.7: Microbenchmarking Results for reduction in XMP and MPI
Chapter 4

Application Benchmarking

4.1 Image Reconstruction

A image processing algorithm has been selected to implement for application benchmarking for this project. The MPI version of the algorithm was already developed before this project, and the XMP implementation of the algorithm has been developed during this project. Both XMP and MPI versions have been benchmarked and analysed.

4.1.1 Algorithm

The selected image processing algorithm was borrowed from a technical report of HPCx. This algorithm takes an image contains edge pixels as input, and reconstruct the image based on the edge pixels. It is the reverse process of a kind of edge detection. Figure 4.1 shows an example of the input and output images of this algorithm.
For an input image with size $M \times N$, the image reconstruction algorithms defines three $M \times N$ size arrays, $edge$, which is used to store the input image pixels, $old$ and $new$, and run for a number of iterations. The $old$ array is used to store the pixel of the previous iteration, while the $new$ array is used to store the pixel of the interim reconstruction results at each iteration. At each iteration, for each pixel $i, j$ in the image, the algorithm calculate the value of the pixel in the $new$ array following the equation at figure 4.2 When all pixels in $new$ have been calculated at one iteration, the pixels in $new$ is copied to $old$ for the next iteration.

$$new_{i,j} = \frac{1}{4}(old_{i-1,j} + old_{i+1,j} + old_{i,j+1} + old_{i,j-1} - edge_{i,j})$$

Figure 4.2: Equation of the image reconstruction for the pixel $i, j$ in the image

### 4.1.2 Implementations

**Decomposition**

Both the XMP and MPI implementation of the image reconstruction algorithm use the two-dimensional approach to decompose the image data. In the XMP implemen-
tation, the decomposition of the image data is done by declaring the image as a two-dimensional global array and aligning it to the node set by the `align` directive. A code fragment has been provided at listing 4.1. In the MPI implementation, for size of image being \( M \times N \) and number of processes being \( p \), the decomposition is implemented by allocating a size \( \frac{M}{p} \times \frac{N}{p} \) chuck of memory on each process.

Listing 4.1: Code fragment of global arrays declaration for image data decomposition in the XMP implementation

```c
/* Declare node set and template */
#pragma xmp nodes p(ny_p, nx_p)
#pragma xmp template t(0:ny_t-1, 0:nx_t-1)
#pragma xmp distribute t(block, block) onto p
/* Declare global array of image data */
float img[NX][NY];
#pragma xmp align img[i][j] with t(j, i)
float old[NX][NY];
#pragma xmp align old[i][j] with t(j, i)
#pragma xmp shadow old[1:1][1:1]
float new[NX][NY];
#pragma xmp align new[i][j] with t(j, i)

/* Special treatment for the limitation of
* the current XMP compiler */
#pragma xmp template s(0:ny_t*ny_p-1, 0:nx_t*nx_p-1)
#pragma xmp distribute s(block, block) onto p
float img_buf[NX*NX_P][NY*NY_P];
#pragma xmp align img_buf[i][j] with s(j, i)
```

Scattering and Gathering

In both implementations of the algorithms, the image data is read by the first process, scattered from that process to all processes in the program, gathered from the processes to the first process and finally written by the first process. In the XMP implementation, the image scattering and gathering are handled by the `gmove` directive. Since the current release of the XMP compiler (0.5.4-1) does not support `gmove` for assignments from global data to local data, a dummy global array has been declared to store the image at its first local section, which is on the first node. The code fragment of this procedure is shown in listing 4.2 and 4.3. In the MPI implementation, the scattering and gathering are done by `MPI_Send` and `MPI_Recv`. Alternatively, this can be done with `MPI_Scatter` and `MPI_Gather` with user-defined MPI data type. The MPI code fragment is shown in listing 4.4 and listing 4.5.

Listing 4.2: Code fragment of image scattering in the XMP implementation
```c
#pragma xmp task on p(1, 1)
{
    printf("Scattering image ...
");
#pragma xmp gmove
    img_buf[0:NX-1][0:NY-1] = img_buf_m[0:NX-1][0:NY-1];
}
#pragma xmp gmove
img[0:NX-1][0:NY-1] = img_buf[0:NX-1][0:NY-1];
```

Listing 4.3: Code fragment of image gathering in the XMP implementation

```c
#pragma xmp gmove in
img_buf[0:NX-1][0:NY-1] = img[0:NX-1][0:NY-1];
#pragma xmp task on p(1, 1)
{
    for (i=0; i<nx; i++)
    {
        for (j=0; j<ny; j++)
        {
            img_buf_m[i][j] = img_buf[i][j];
        }
    }
}
```

Listing 4.4: Code fragment of image scattering in the MPI implementation

```c
if (rank == 0)
{
    /* Send image decomposition to the rest of ranks */
    for (i=0; i<size_x; i++)
    {
        for (j=0; j<size_y; j++)
        {
            if (i*size_y+j != 0)
            {
                for (row=1; row<nxp+1; row++)
                {
                    for (col=1; col<nyp+1; col++)
                    {
                        edge[row][col] = masterbuf[i*nxp+row-1][j*nyp+col-1];
                    }
                }
                MPI_Send(&edge[0][0], ...
```
for (row=1; row<nxp+1; row++)
{
    for (col=1; col<nyp+1; col++)
    {
        edge[row][col] =
            masterbuf[row-1][col-1];
    }
}
else
{
    MPI_Recv(& edge[0][0], ...
}

Listing 4.5: Code fragment of image gathering in the MPI implementation

if (rank== 0)
{
    for (i=0; i<size_x; i++)
    {
        for (j=0; j<size_y; j++)
        {
            if (i*size_y+j!=0)
            {
                MPI_Recv(& old[0][0], ...
            }
        for (row=0; row<nxp && i*nxp+row<nx; row++)
        {
            for (col=0; col<nyp && j*nyp+col<ny; col++)
            {
                masterbuf[i*(nx/size_x)+row][j*(ny/size_y)+col]= old[row+1][col+1];
            }
        }
    }
}
4.1.3 Halo Swapping and Image Reconstruction

The parallel image reconstruction is implemented with the loop directive in the XMP implementation. Since the algorithm accesses the neighbouring pixels in the old array of each index pair \( i, j \), each process of the program needs to have access to the image pixels at the boundaries of the decompositions on its neighbouring processes. This kind of image pixels are referred as halo data, and each process needs to swap halo data with its four non-diagonal neighbours before starting processing the pixels at each iteration. In the XMP implementation of the algorithm, the halo swapping is handled with the reflect directive. The code fragment is shown in listing 4.6. In the MPI implementation, it is done by \texttt{MPI\_Send} and \texttt{MPI\_Recv}. The code fragment of halo swapping in the MPI version of image processing is provide in listing 4.7.

Listing 4.6: Code fragment of halo swapping and image processing in the XMP implementation

```c
for (i = 0; i < max_iter; i++)
{
    #pragma xmp reflect (old)
    #pragma xmp loop (col, row) on t(col, row)
    for (row = 0; row < nx; row++)
    {
        for (col = 0; col < ny; col++)
        {
            float top, bottom, left, right;
            top = old[row - 1][col];
            bottom = old[row + 1][col];
            left = old[row][col - 1];
            right = old[row][col + 1];
            new[row][col] = (top + bottom + left +
                            right - vbuf[row][col]) * 0.25;
        }
    }
    #pragma xmp loop (col, row) on t(col, row)
    for (row = 0; row < nx; row++)
    {
        for (col = 0; col < ny; col++)
        {
            old[row][col] = new[row][col];
        }
    }
}
```

Listing 4.7: Code fragment of halo swapping in the MPI implementation

```c
for (row = 1; row < nvp + 1; row++)
```


```c
{
    halo_buf_left_send[row - 1] = old[row][1];
    halo_buf_right_send[row - 1] = old[row][nyp];
}

/* Swap halos with standard send */
if (0 == coords[0] % 2)
{
    MPI_Send(&old[1][1], ...)
    MPI_Send(&old[nxp][1], ...)
    MPI_Recv(&old[0][1], ...)
    MPI_Recv(&old[nxp + 1][1], ...)
}
else
{
    MPI_Recv(&old[0][1], ...)
    MPI_Recv(&old[nxp + 1][1], ...)
    MPI_Send(&old[1][1], ...)
    MPI_Send(&old[nxp][1], ...)
}
if (0 == coords[1] % 2)
{
    MPI_Send(halo_buf_left_send, ...)
    MPI_Send(halo_buf_right_send, ...)
    MPI_Recv(halo_buf_left_recv, ...)
    MPI_Recv(halo_buf_right_recv, ...)
}
else
{
    MPI_Recv(halo_buf_left_recv, ...)
    MPI_Recv(halo_buf_right_recv, ...)
    MPI_Send(halo_buf_left_send, ...)
    MPI_Send(halo_buf_right_send, ...)
}
for (row = 1; row < nxp + 1; row++)
{
    old[row][0] = halo_buf_left_recv[row - 1];
    old[row][nyp + 1] = halo_buf_right_recv[row - 1];
}

33
```
4.1.4 Results

Both the XMP and the MPI versions of the image reconstruction program have been benchmarked on HECToR with number of processes doubles from 32 to 2048. The runtime of these parts of the programs have been measured: image data scattering and gather, halo swapping and image processing. A $2048 \times 2048$ size image has been used as the input, and the reconstruction programs have been run for 5000 iterations to reconstruct the image.

Scattering and Gathering

Figure 4.3 and figure 4.4 provide runtime results of image scattering and gathering in both the XMP version and the MPI version of the image reconstruction programs. The results show a great difference between the performance of the image scattering implementation in XMP and the one in MPI. While the runtime of the MPI image scattering and gathering remained less than 2 seconds, the runtime of the XMP implementation grew linearly from about 14 to more than 220 seconds when the number of processes increased from 32 to 2048. This finding is consistent with the microbenchmarking results comparing the `gmove` directive and `MPI_Send` and `MPI_Recv` which is shown in figure 3.1.
Halo Swapping

Figure 4.5 shows the runtime results of halo swapping in the image reconstruction programs. For the XMP version, the runtime of halo swapping fluctuated when the number of processes increased from 32 to 512, and kept decreasing from about 0.91 seconds to about 0.32 seconds when the number of processes increases from 512 to 2048. For the MPI version, the runtime decreased significantly from about 3.76 seconds to 1.36 seconds when the number of processes increased from 32 to 128, and bounced up slightly when the number of processes reached 256. The runtime kept gradually decreasing when the number of processes grew from 256 to 2048. By comparing the runtime of the XMP version with the runtime of the MPI version, it has been revealed in the results that the XMP version of image data halo swapping has a better performance than the MPI version does. This finding is different with the finding of the one-dimensional halo swapping microbenchmarking results.

Reconstruction

Figure 4.6 shows the runtime results of parallel image reconstruction in both the XMP and MPI programs. The image reconstruction runtime of both the XMP and the MPI versions decreased in a logarithmic manner from about 88 seconds to about 2 second when the number of processes increased from 32 to 2048. The difference between the runtime of the two programs is negligible. This finding is consistent with the microbenchmarking results of the parallel loop overhead measuring.
Figure 4.5: Runtime result of halo swapping in the XMP and the MPI image processing programs

Figure 4.6: Runtime result of image reconstruction in the XMP and the MPI image processing programs
Chapter 5

Conclusion and Future Work

This project has investigated some aspects of the performance of XcalableMP by developing synthetic and application benchmarks. The synthetic benchmarking (microbenchmarking) has measured the performance of a set of XMP directives when running on from 2 to 2048 number of nodes on HECToR. The application benchmarking has been developed by implementing an image processing algorithm which involves halo swapping in XMP. The runtime of image scattering and gathering, image reconstruction and halo swapping steps in the XMP image processing program has been measured when running on from 2 to 2048 number of nodes on HECToR. Time measurement has also been done for the MPI equivalent implementations of XMP programs during both synthetic and application benchmarking, and the performance of the XMP implementations and the MPI implementations has been compared during the two benchmarking processes.

The results of the synthetic benchmarks has revealed that the performance of some XMP directives, including 
\textit{gmove}, \textit{reflect in one-dimension}, \textit{barrier}, \textit{bcast} and \textit{reduce}, was significantly worse than the performance of their equivalent MPI benchmarking implementations. The only directive which was able to catch up with the performance of its equivalent MPI implementation was the \textit{loop} directive.

The results of the application benchmark has revealed that the MPI implementation of image scattering and gathering greatly outperformed the equivalent XMP implementation. The runtime results of parallel image reconstruction of the XMP and the MPI implementations were close. Finally, the performance of halo swapping of the XMP implementation was slightly better than the performance of the equivalent MPI implementation.

Despite the fact that the performance of several XMP benchmarking programs were significantly worse than the performance of their equivalent implementations in MPI,
the project is unable to confidently state that the parallel code generated by the XMP compiler is slower than the MPI equivalent code in general, due to the factors which might have interfered the benchmarking processes. As mentioned in the background chapter, the directives involved in communication were translated to one-sided communication procedures in MPI, while the MPI benchmarking programs in this project were mostly written with standard send and receive, because of the limited time of this project. Consequently the performance difference of one-sided communication and standard send and receive might have significantly interfered the evaluation of the performance of the code generated by the XMP compiler. Also, for some of the benchmarking XMP programs of this project, there are alternative MPI implementations which could have been developed and compared against the XMP implementation. For example, for swapping image halos of the 2D decomposition in the application benchmark, instead of moving non-contiguous halo data between the image array and the halo buffer, one can use MPI data type to swap halo data, which is possible to have a significantly different performance.

More benchmarking of XMP can be done in the future when more features are supported by newer versions of the XMP compiler. As some limitations of this project have been stated above, one is encouraged to overcome these limitations in the future work. The future projects are advised to be more aware about the performance of the one-sided communication on the platform before evaluating the performance of the code generated by the XMP compiler. It is also advised to investigate different implementations of the same benchmark when writing code in other programming models to compared with the XMP code. Lastly, since some other one-sided communication implementations, such as Cray SHMEM, are known to have a better performance than the MPI one-sided communication on HECToR [14], it is possible to achieve better performance of XMP code in the future by coupling the XMP compiler with the one-sided communication implementations which has a better performance.
Chapter 6

Bibliography


