A Performance Evaluation of MPI Shared Memory Programming

DAVID KARLBOM
A Performance Evaluation of MPI Shared Memory Programming

DAVID KARLBOM

Master’s Thesis at CSC
Supervisor: Stefano Markidis & Roman Iakymchuk
Examiner: Erwin Laure

June 16, 2016
Abstract

The thesis investigates the Message Passing Interface (MPI) support for shared memory programming on modern hardware architecture with multiple Non-Uniform Memory Access (NUMA) domains. We investigate its performance in two case studies: the matrix-matrix multiplication and Conway’s game of life. We compare MPI shared memory performance in terms of execution time and memory consumption with the performance of implementations using OpenMP and MPI point-to-point communication, also called "MPI two-sided". We perform strong scaling tests in both test cases.

We observe that MPI two-sided implementation is 21% and 18% faster than the MPI shared and OpenMP implementations respectively in the matrix-matrix multiplication when using 32 processes. MPI shared uses less memory space: when compared to MPI two-sided, MPI shared uses 45% less memory. In the Conway’s game of life, we find that MPI two-sided implementation is 10% and 82% faster than the MPI shared and OpenMP implementations respectively when using 32 processes. We also observe that not mapping virtual memory to a specific NUMA domain can lead to an increment in execution time of 64% when using 32 processes. The use of MPI shared is viable for intranode communication on modern hardware architecture with multiple NUMA domains.
Referat

En utvärdering av *MPI shared memory* -
programmering med inriktning på prestanda


Vi observerar att MPI-two sided är 21% snabbare än MPI shared och 18% snabbare än OpenMP för matris-matris multiplikation när 32 processorer användes. För samma testdata har MPI shared en 45% lägre minnesförburkning än MPI two-sided. För Conway’s game of life är MPI two-sided 10% snabbare än MPI shared samt 82% snabbare än OpenMP implementation vid användandet av 32 processorer. Vi kunde också utskilja att om ingen mappning av virtuella minnet till en specifik NUMA domän görs, leder det till en ökning av exekveringstiden med upp till 64% när 32 processorer används. Vi kom fram till att MPI shared är användbart för intranode kommunikation på modern hårdvaruarkitektur med flera NUMA domäner.
Acknowledgement

Writing a thesis can be a long and at times a very straining task. I would, therefore, like to thank the people that helped me during the process of researching and writing this thesis.

First, I would like to thank PDC Center for High Performance Computing at KTH Royal Institute of Technology in Stockholm for letting me conduct this master thesis and utilize their facilities and computers. In particular, I want to thank my supervisors Stefano Markidis and Roman Iakymchuk for pushing me to make the thesis what it is today. I would also like to thank my family and friends for helping me to proofread the report.
4 Benchmark Applications 23

4.1 Matrix-Matrix Multiplication ........................................... 23
   4.1.1 MPI Shared ......................................................... 24
   4.1.2 MPI Two-Sided .................................................... 25
   4.1.3 OpenMP .......................................................... 25

4.2 Conway’s Game of Life .................................................. 26
   4.2.1 MPI Shared ......................................................... 29
   4.2.2 MPI Shared Non-Contiguous .................................... 30
   4.2.3 MPI Two-Sided .................................................... 31
   4.2.4 OpenMP .......................................................... 33

4.3 Benchmark Environment ................................................ 33
   4.3.1 Beskow ........................................................... 33
   4.3.2 Tegner ............................................................ 34

4.4 Simulation Set-Up ........................................................ 35
   4.4.1 Matrix-Matrix Multiplication .................................... 35
   4.4.2 Conway’s Game of Life ......................................... 35

4.5 Performance Measurements ............................................ 36
   4.5.1 Code Profilers .................................................... 37

4.6 Verification Tests ........................................................ 38
   4.6.1 Matrix-Matrix Multiplication .................................... 38
   4.6.2 Conway’s Game of Life ......................................... 39

5 Results 41

5.1 Verification Results ...................................................... 41
   5.1.1 Matrix-Matrix Multiplication .................................... 41
   5.1.2 Conway’s Game of Life ......................................... 42

5.2 Performance Results ..................................................... 43
   5.2.1 Matrix-Matrix Multiplication .................................... 43
   5.2.2 Conway’s Game of Life ......................................... 44

5.3 Comparison of Compilers Interaction with MPI Shared ......... 47

5.4 Performance Comparison of MPI Shared, MPI Two-Sided and OpenMP .......................... 49
   5.4.1 Matrix-Matrix Multiplication .................................... 49
   5.4.2 Conway’s Game of Life ......................................... 51

6 Analysis and Conclusion 57

6.1 Analysis ................................................................. 57
   6.1.1 Allocation of memory ............................................. 57
   6.1.2 Matrix-Matrix Multiplication .................................... 57
   6.1.3 Conway’s Game of Life ......................................... 58
   6.1.4 MPI Shared vs OpenMP ......................................... 58
   6.1.5 MPI Shared vs MPI Two-Sided .................................. 59

6.2 Ethics, Social Aspects and Sustainability ......................... 59

6.3 Conclusion .............................................................. 59
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Table of the different datasets and their sizes, which is used for evaluating matrix-matrix multiplication.</td>
<td>35</td>
</tr>
<tr>
<td>4.2</td>
<td>Table of the different datasets and their sizes, which is used for evaluating the Conway’s game of life.</td>
<td>36</td>
</tr>
<tr>
<td>5.1</td>
<td>The mean time, variance, speedup and efficiency for the MPI shared implementation of matrix-matrix multiplication executed on Beskow. <em>Mmult_big</em> was used as dataset.</td>
<td>44</td>
</tr>
<tr>
<td>5.2</td>
<td>The mean time, variance, speedup and efficiency for the MPI shared implementation of Conway’s game of life executed on Beskow. <em>Strong_b</em> was used as dataset.</td>
<td>46</td>
</tr>
<tr>
<td>5.3</td>
<td>The time (in seconds) it takes to complete one round of the Conway’s game of life with different compilers when using malloc or MPI shared memory.</td>
<td>47</td>
</tr>
<tr>
<td>5.4</td>
<td>The mean time, variance, speedup and efficiency for MPI two-sided, MPI shared and OpenMP implementations of matrix-matrix multiplication. <em>Mmult_big</em> was used as the dataset and the tests were run on Beskow.</td>
<td>50</td>
</tr>
<tr>
<td>5.5</td>
<td>The memory consumption of the matrix-matrix multiplication application per process element, provided by CrayPat.</td>
<td>50</td>
</tr>
<tr>
<td>5.6</td>
<td>The speedup of MPI two-sided, MPI shared, OpenMP, and MPI shared without initializing the memory, for Conway’s game of life.</td>
<td>51</td>
</tr>
<tr>
<td>5.7</td>
<td>The mean time, variance, speedup and efficiency for MPI two-sided, MPI shared and OpenMP implementations of Conway’s game of life. <em>Strong_b</em> was used as the dataset and the tests were run on Tegner.</td>
<td>52</td>
</tr>
<tr>
<td>5.8</td>
<td>The weak and strong scaling efficiency for Conway’s game of life that was produced by MPI shared and MPI two-sided.</td>
<td>53</td>
</tr>
<tr>
<td>5.9</td>
<td>The overhead and High Memory Water Mark from CrayPat for MPI shared and MPI two-sided when using 32 PE on Conway’s game of life.</td>
<td>54</td>
</tr>
<tr>
<td>5.10</td>
<td>The mean time, variance, speedup and efficiency for MPI two-sided and MPI Shared implementations of Conway’s game of life. <em>Strong_t</em> was used as dataset and the tests were run on Tegner.</td>
<td>55</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>3.1</td>
<td>A NUMA architecture with two CPUs. The connection between the two NUMA nodes is provided by the network.</td>
<td>10</td>
</tr>
<tr>
<td>3.2</td>
<td>Code Snippet 3.1’s pinning of the processing element to cores. Both NUMA nodes have two cores and their own local memory area.</td>
<td>12</td>
</tr>
<tr>
<td>3.3</td>
<td>The two ways of domain decomposition: the one-dimensional is presented on the left and the two-dimensional is presented on the right.</td>
<td>14</td>
</tr>
<tr>
<td>3.4</td>
<td>This figure shows the different delays when using blocking communication. The blue bold part of the path is idle time.</td>
<td>15</td>
</tr>
<tr>
<td>3.5</td>
<td>The two MPI memory models for remote memory windows. The separated model is presented on the left and the unified model is presented on the right.</td>
<td>16</td>
</tr>
<tr>
<td>3.6</td>
<td>Two examples of synchronizations call. Active target synchronizations are present on the left. Passive target synchronizations are presented on the right.</td>
<td>17</td>
</tr>
<tr>
<td>3.7</td>
<td>Window contiguous allocation. The black parts are the starting allocation for a PE. PE 2 and PE 3 allocate different memory sizes.</td>
<td>19</td>
</tr>
<tr>
<td>3.8</td>
<td>The 'initial' pointer points to the start of the respective window. 'PROCESS_NULL' is the pointer from MPI_Win_share_query with rank MPI_PROC_NULL. 'Other' is the pointer from MPI_Win_share_query with the other processes rank.</td>
<td>19</td>
</tr>
<tr>
<td>4.1</td>
<td>The figure presents two different types of stencil operations. The red middle point is the current position and the surrounding blue points are the positions used for the stencil operation. A 4-point stencil is presented on the left and an 8-point stencil is presented on the right.</td>
<td>26</td>
</tr>
<tr>
<td>4.2</td>
<td>The Figure represents the two different approaches of transformation in an 8 × 8 grid. The red area represents current position, blue represents the stencil and grey represents the non-evaluated positions. The gray area is only applicable in the fixed transformation on the left. The torus transformation is presented on the right.</td>
<td>27</td>
</tr>
<tr>
<td>4.3</td>
<td>Allocation of the MPI shared contiguous memory for Conway’s game of life. The yellow part of the memory allocation is the old grid and the blue part is the new grid.</td>
<td>30</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

4.4 Allocation of the MPI non-contiguous shared memory for Conway’s game of life. The yellow part of the memory allocation by PE 0 and the blue part is allocated by PE 1. 31

4.5 Ghost cells for a $8 \times 8$ matrix with two PEs. Each PE has a ghost row which is the other PE’s boundary row. The orange part is the ghost cell from PE 0 to PE 1 and the purple is from PE 1 to PE 0. 32

4.6 A Cray XC40 CPU node. The red arrows represent the connections to the memory. 34

4.7 Matrix multiplication. The matrices A and B are represented as mmult_mini and use as input matrices. The C matrix is the output matrix. 38

4.8 Snapshot of the initial state (left) and after 1201 rounds (right) of Conway's game of life with the stencil_mini dataset. 39

5.1 The output from the mmult_mini test set when evaluated with different implementations of the matrix-matrix multiplication. The output of the serial version is presented on the left and the output of the MPI shared memory implementation is presented on the right. 41

5.2 Output from the MPI shared implementation of Conway’s game of life is represented on the right side. The left grid is the output from the serial program, which is the control instance. 42

5.3 Speedup measurement (left) and strong scaling (right) of matrix-matrix multiplication with MPI shared windows on Beskow. 43

5.4 Speedup measurement (left) and strong scaling measurement (right) for the two MPI shared memory implementations of Conway’s game of life with initialization. ”MPI Shared-No-Contiguous” is the implementation using no contiguous memory. ”MPI Shared-Contiguous” is the contiguous implementation. 44

5.5 Speedup measurement (left) and strong efficiency (right) showing the impact of initializing the MPI shared memory region before using it on the NUMA architecture. 45

5.6 Process affinities effect on performance seen in speedup (left) and strong scaling (right). ”CPU” places the PEs in ascending order meanwhile ”Half” distributes the PEs evenly between the NUMA nodes. 46

5.7 The speedup measurements (left) and strong efficiency measurements (right) for MPI two-sided and MPI shared memory, when compiled with GCC compiler on Beskow. 47

5.8 Compiler performance difference between Intel and GCC compilers when using MPI shared on Tegner system. Speedup on the left side and strong scaling on the right side. 48

5.9 Speedup measurements on the left side and strong efficiency on the right side for the implementations of MPI two-sided, MPI shared and OpenMP with the application matrix-matrix multiplication on Beskow. 49

5.10 Total memory usage summed over all processing element. CrayPat’s High Memory Water Mark has been used for the measurement. 51
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.11</td>
<td>Speedup measurement (left) and strong scaling measurement (right) for MPI two-sided, MPI shared and OpenMP implementations of Conway’s game of life.</td>
</tr>
<tr>
<td>5.12</td>
<td>Weak scaling measurement for MPI two-sided, MPI shared memory and OpenMP implementation of Conway’s game of life on Beskow.</td>
</tr>
<tr>
<td>5.13</td>
<td>Pie chart taken from appearance2 CrayPat to representing the run time overhead from MPI two-sided and MPI shared when using 32 PE on Conway’s game of life.</td>
</tr>
<tr>
<td>5.14</td>
<td>Speedup (left) and strong scaling (right) for the Conway’s game of life using MPI two-sided and MPI shared memory on Tegner.</td>
</tr>
</tbody>
</table>

### List of Code Snippets

<table>
<thead>
<tr>
<th>3.1</th>
<th>Example on how to use Cray Application Level Placement Schedule to pin processing elements to physical cores.</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2</td>
<td>A simple program of shared memory programming with MPI shared memory windows.</td>
</tr>
<tr>
<td>4.1</td>
<td>The simple matrix-matrix multiplication code with transpose of the B matrix.</td>
</tr>
<tr>
<td>4.2</td>
<td>Pseudocode of matrix-matrix multiplication with MPI shared.</td>
</tr>
<tr>
<td>4.3</td>
<td>Pseudocode of matrix-matrix multiplication code with OpenMP.</td>
</tr>
<tr>
<td>4.4</td>
<td>Code for Conway’s game of life stencil operation.</td>
</tr>
<tr>
<td>4.5</td>
<td>Implementation of Conway’s game of life with MPI contiguously shared memory.</td>
</tr>
<tr>
<td>4.6</td>
<td>Implementation of Conway’s game of life with MPI shared non-contiguously memory.</td>
</tr>
<tr>
<td>4.7</td>
<td>Implementation of Conway’s game of life with MPI two-sided by using ghost cells.</td>
</tr>
<tr>
<td>4.8</td>
<td>The work flow how to use the profiler CrayPat on a cray systems.</td>
</tr>
<tr>
<td>4.9</td>
<td>Example of how to use the diff command.</td>
</tr>
</tbody>
</table>
## Acronym

**API**  Application Programming Interface.  \[^{5}\]

**CPU**  Central Processing Unit.  \[^{2,9,11,13}\]

**HPC**  High Performance Computing or High Performance Computer.  \[^{10,11,33,58}\]

**MPI**  Message Passing Interface.  \[^{2,7,9,14,19,21,23,25,29,33,35,36,38,44,46,47,49,55,58,61}\]

**NUMA**  Non-Unified Memory Access.  \[^{10,13,19,24,29,44,49,57,58}\]

**OpenMP**  Open Multi-Processing.  \[^{2,4,6,23,25,33,36,38,49,51,58,60}\]

**PE**  Processing Element.  \[^{2,5,10,15,17,22,24,25,30,32,36,37,41,42,44,48,53,57,59}\]

**PGAS**  Partitioned Global Address Space.  \[^{7}\]

**QPI**  Intel QuickPath Interconnect.  \[^{10,33}\]

**RDMA**  Remote Direct Memory Access.  \[^{3}\]

**RMA**  Remote Memory Access.  \[^{2,3,6,7,16,17}\]

**SMP**  Symmetric Multiprocessing.  \[^{9}\]

**TLB**  Translation Lookaside Buffer.  \[^{6}\]

**UMA**  Unified Memory Access.  \[^{9,10,12}\]

**UPC**  Unified Parallel C.  \[^{5}\]
Chapter 1

Introduction

In this chapter, we present the outline of the thesis as well as the goals and the problem statements. Before this we present an introduction to high performance computing (HPC), shared memory programming as well as the Message Passing Interface standard.

1.1 High Performance Computing

Over the last 60 years, high performance computers have been used, for example, to make weather predictions, run physics simulations and perform medical research. Over decades both hardware and software have continuously improved, creating better possibilities for higher performance. But it was not until the 90’s that the number of cores started to increase rapidly. At present, the main part of the high performance computers are clusters or massively parallel processing machines [7]. Both of these types of machines have similar architecture. Processors are connected on a computation node and multiple nodes are connected by a network. To have a good knowledge of the underlying hardware architecture is of the utmost importance to generate the highest performance. Furthermore, how the software interacts with this specific hardware configuration has also an impact on performance. Even the slightest difference could have impact the performance. For example, the configuration of core per node, cache sizes and even the location of the nodes in the network can have a huge impact on the performance. The fastest high performance computers consist of more than 50,000 cores and have the capability of solving enormous problems in a few minutes [7]. This could have taken days to solve using a personal computer.

1.2 Shared Memory Computing Systems

For almost 40 years Moore’s law has been pushing the big hardware manufacturers to create smaller transistors and to place an increasing amount of transistors per
CHAPTER 1. INTRODUCTION

chip. With increased numbers of transistors, the hardware solutions will be faster. However, currently the time to double the transistors per area is increasing.

To meet the demand for faster computers, current computer system uses more than a single Central Processing Unit (CPU) per chip. Modern systems went from having dual core multiprocessors to the state-of-the-art 64 core multiprocessors. To utilize all of these cores, software developers need to work with some parallel programming model. There are among others the message passing model, the shared memory model, and the remote memory model. When the main memory of the computer is available to all cores the most commonly used memory model is the shared memory model. Furthermore, it is not an easy task to create both a fast and low memory consuming parallel program. Good libraries and intuitive Application Programming Interfaces (API) are needed. Some of the most used APIs uses an implementation of threads, which uses the shared memory model. For example Open Multi-Processing (OpenMP) and POSIX-threads in which the fork and join model is used. The fork and join model creates parallelism by having parts of the code running in a separate thread [27]. In 2013, the Message Passing Interface (MPI) committee released a new standard which includes the possibility of shared memory programming using a shared memory model [30].

1.3 The MPI Standard

The MPI was created in 1992 as an effort to extract the best features of already existing message passing systems and create a standardized way to use message passing on different computer systems [5]. Before MPI, there were implementations of the message passing model which were both fast and efficient, but they were almost exclusively featured with vendor-specific calls. It would take both time and effort to rewrite all the software to adapt to any other hardware setup. With the MPI standard, vendors and researchers can implement their own implementation of the MPI standard in the form of a library. With a standardized API, the developers’ applications can easily be adapted to any implementation of the standard and therefore applications become system independent. The early stages of the standard were developed around sending point-to-point messages to utilize the message passing model. A point-to-point message is a two-way message exchange from one Processing Element (PE) to another, where both of the Processing Elements are involved [36]. Point-to-point communication is also known as two-sided communication. In 1998, the standard was updated to MPI-2. With the new standard, MPI was not only supporting the message passing model, the standard also included the Remote Memory Access (RMA) model [30]. When using the RMA model the targeting PE does not need to be involved in the communication. As a result, the developers’ have more freedom when programming. RMA communication is more commonly known as one-sided communication [30] [20]. After more than 10 years the standard was updated again (MPI-3) to adapt to the modern hardware

\[^{1}\text{Processing Element} \text{ is a collective term to describe a rank in MPI or a thread in OpenMP.}\]
architecture. The message passing part was left intact but the RMA part of the standard was improved to also support Remote Direct Memory Accesses (RDMA). With these changes, the opportunity came to also support a shared memory model. If a shared memory model is included in the standard, developers can use both the message passing model and the shared memory model at the same time to create a hybrid model. The shared memory model is based upon creating memory regions which are shared between processes [5].

In this thesis, we will evaluate MPI shared memory model.

1.4 Goals of the Thesis

In this thesis, we aim at evaluating the support MPI has for shared memory programming on modern hardware architecture. To be more specific, the goals of the thesis are:

- To evaluate how to use MPI shared memory windows.
- To evaluate the scaling performance of MPI shared.
- To evaluate the memory consumption of MPI shared.
- To evaluate how MPI shared matches against currently used parallel APIs.

When evaluating these four goals there is a need for applications. Two applications matrix-matrix multiplication and a stencil application will be used to evaluate the goals of the thesis. All of these goals will be conquered to answer the question: Does MPI have an applicable possibility of utilizing a shared memory model?

1.5 Contribution

The main contributions of this work are:

- The thesis provides an in-depth analysis of MPI support for shared memory programming. The MPI support for shared memory programming has been recently added to the MPI standard so very few articles have been published on this topic. This work provides one of the first introductions to MPI shared memory programming.
- The thesis provides for the first time (to Author’s knowledge) a performance comparison of MPI shared programming with OpenMP programming.
- The thesis presents for the first time an analysis of the memory consumption of MPI shared memory programming.
1.6 Overall Structure of Thesis

In the second chapter, we present previous research in shared memory programming. The evaluated shared memory models are OpenMP and MPI shared memory models. In the third chapter, different memory models and how parallelization of a serial program can be performed are described. Also, software interaction with MPI two-sided and MPI shared memory is presented. In the fourth chapter, the experiments and the environment used in the experiment are described. In the fifth chapter, the results are presented. In the sixth chapter, conclusions are drawn and the results are discussed.
Chapter 2

Literature Study

In this chapter, a literature study of previous work of utilizing the shared memory models with OpenMP and MPI is presented. Also problems when using the shared memory models and the message passing model together are presented.

2.1 Programming Shared Memory Systems

In this section, we present previous work performed with OpenMP, one of the most commonly used APIs for shared memory programming, and MPI shared memory programming.

2.1.1 OpenMP

How to create parallelism with OpenMP is presented in the book [12]. A comprehensive evaluation of a matrix-vector multiplication application is presented. The best way to utilize the parallel model is presented in both FORTRAN and C. The book [12] also outlines an implementation of a stencil application. While the book presents how to implement OpenMP programs, there is no evaluation of performance against any other standard or API. On the other hand, the scientific paper [31] presents an evaluation between how profitable in terms of performance OpenMP is compared to both MPI two-sided and Unified Parallel C (UPC). Ref. [31] also presents that the application with OpenMP is better on shared memory systems, due to not copying data between the PEs. Although when going from only a shared memory model into a distributed memory model, OpenMP cannot any longer take advantages of the increased number of cores. The two applications evaluated are matrix-matrix multiplication and a Sobel kernel function. Implementation details on how to create applications and how to modify an existing serial program into a parallel program with OpenMP is described in both [12] and [10]. On the other hand, in [33] the authors describes the pitfalls in the OpenMP models when sleeping and awakening threads. The report also presents that some compiler optimization will not be performed when the program is compiled with the OpenMP library. All
of these reports present that it is not as easy as adding the parallel directives to create good parallelism. As can be seen, there has been a lot of previous work in regards to OpenMP and MPI two-sided but no evaluation of OpenMP against MPI shared.

2.1.2 MPI Shared Memory

MPI shared memory programming is still very new. The standard was adopted in 2013 by the MPI Forum [5]. The MPI Forum were questioning how to add a true zero-copy communication to the standard. They came to the conclusion that the best way to include a shared memory model was to extend the RMA part of the standard. In the work [24] and [25] the extension to the Remote Memory Access part of the standard is presented. Also, the new functionality of creating and using shared memory is presented. Both the scientific papers start by presenting how to exposing the memory to all processes to finishing with how to access other processes shared memory regions. The papers also include a higher level presentation of how the standard is implemented in MPICH and OpenMPI.

The implementation of MPICH is evaluated to overcome some of the performance bottlenecks when transforming from a two-sided application to a shared application in [44]. With the implementation of a five-point stencil, the authors are conquering a compiler vectorization problem as well as a problem with an increased number of page faults.

The vectorization problem occurred when the pointer produced by MPI shared does not have the same properties as the pointer produced by malloc. This can be easily solved with an alias pointer or using the restricted keyword. The other bottleneck brought up and solved is the increased amount of page faults from MPI shared memory. The solution presented is to modify the Linux systems environment to use huge pages which the MPI two-sided implementation does by default. With these changes the program has the same property as an MPI two-sided implementation in the page faults and TLB misses.

In both [24] and [44] the communications overhead is presented for the implementation of their five-point stencil applications. Each experiment uses only two processes. Both reports showcase the difference in communication time between the implementation of MPI two-sided (single copy) and MPI shared memory. In addition [44] has also evaluated the communication time of an MPI two-sided (double copy) implementation. In the implementation blocking calls of MPI two-sided are used. The authors report a decrement in the communication overhead of 30-80%, depending on the system.

A halo exchange is evaluated in comparison with MPI two-sided using a modified MPPTest in [9]. In Ref. [44] the result is presented for only two cores. The authors of [9] presents the results for a range of different number cores and the performance of the application is measured. The authors define the overhead to be the synchronization instead of the regular communication overhead for the two-sided communication.
2.2 Programming Using Hybrid Memory Model

The RMA functionality has been used for creating an implementation of the PGAS standard. At the time when Ref. [43] was written the shared memory segment was not included. In Ref. [42] the implementation of the same PGAS standard is now using MPI shared for the node’s local communication to bypass the MPI library, to improve the speedup when communicating inside of the node. With the improvements from Ref. [43] to Ref. [42] the library is now compatible with state-of-the-art implementation of PGAS.

These are the previous work which has been evaluated with MPI shared memory programming before the thesis was written. There has been a very low amount of actual performance evaluation. Also, no data about the memory consumption has been presented in previous work. Instead of using communication overhead we will present the performance in a way of speedup. More information about the measurements will be presented in section 4.5.

2.2 Programming Using Hybrid Memory Model

A hybrid model is created when two different memory models are used at the same time. The most commonly used hybrid model combines the message passing model with the shared memory model. In Ref. [33], the authors give the reader a deep explanation of how to use a hybrid model to achieve higher performance. They use the shared memory model for intranode communication and the message passing model for communication between different nodes using the network. Furthermore, [33] address the problem of using threads together with MPI two-sided. To use threads together with MPI becomes increasingly harder developing correct applications.

With the fast increment of cores per node, it is difficult for the network to handle all the communication when all processes are using the network [8]. It can also be seen that allocation of redundant memory for each distinct address space in an pure MPI two-sided approach is not sustainable anymore [28].

Even if the MPI standard supports a mixture of MPI and threads since MPI 2.1 there are issues in implementing a completely thread safe implementation of the standard [21]. In one of the most recent version of MPICH, there are still parts of the library which are not supported in thread safe mode [35].
Chapter 3

Materials and Methods

In this chapter, the applications, method and the foundation for the experiments are presented. In section 3.1 the fundamentals of utilizing the memory hierarchy are described. Section 3.2 describes how parallelization of data intensive algorithms could be implemented. Section 3.3 and 3.4 describe the MPI two-sided and MPI shared memory interfaces.

3.1 Shared Memory Systems

In shared memory all processes use the same memory module and access it through their own hardware port. The memory module later decides if the request can be passed to the memory controller. When creating a serial program, how the virtual memory is mapped to physical memory is usually not important. But for parallel programs the placement of data is of utmost importance, due to the possibility of different accessing times for different placements. This section presents how to utilize different shared memory hardware setups.

3.1.1 Uniform Memory Access Systems

The Unified Memory Access (UMA) system is defined as a system where all memory is accessible through an interconnection network in the same way a single processor accesses its own local memory. The biggest impacting property of a UMA system is that it has a unified memory access time. If the system has the UMA property and all CPUs has the same speed it is called a Symmetric Multiprocessing (SMP) system. When using this memory layout it is easier to do an evaluation of where the data is placed in memory. Programming against a UMA architecture has its advantages, but in larger systems there exist problems when trying to archive a unified access time.
3.1.2 Non-Uniform Memory Access Systems

In the Non-Unified Memory Access (NUMA) system, all the CPUs are attached to the shared memory directly, just like in the UMA system. By having a direct connection to the memory leads to the possibility of addressing every single part of the memory with real addresses. But in NUMA systems, the access time to memory is not unified and is dependent on the location of the data. In a memory bound application, allocating memory in the wrong place can have a huge effect on performance [16]. For instance, if a computer has two CPUs, it is better to place the most used data by PE 0 in PE 0's local domains and vice versa for PE 1. Therefore, placing everything in a single memory domain could lead to a longer execution time due to memory imbalance. On the other hand, this makes it harder to achieve good performance when using a NUMA system than by using a UMA system.

When programming NUMA systems one of the most important properties is to know where the memory actually is allocated onto physical hardware. In a UMA memory architecture, there is no need to think about where the memory is allocated due to unified access time. But larger systems do not generally use a UMA architecture. Instead, large systems use a NUMA architecture. In HPC, there are often several NUMA domains representing a computation node [18].

When accessing data which is located on another NUMA node there is a need to pass the data through an interconnection network. When using this interconnection network there is a transport protocol to send data from one side of the connector to the other side. To transport data over the interconnection network a hardware protocol needs to firstly pack the data, secondly send the packaged data over the interconnection network and lastly unpack the data on the other side. Therefore, this is clearly slower than accessing local memory. Some of these interconnection network are QPI and HyperTransport. In Fig. 3.1 a NUMA system with two different NUMA nodes is presented. The full picture represents a node or a system.

Figure 3.1: A NUMA architecture with two CPUs. The connection between the two NUMA nodes is provided by the network.
3.1. SHARED MEMORY SYSTEMS

When using NUMA system, it is important to know the affinity of the system and how to modify the affinity to achieve good performance. It exists two types of affinity:

- Process affinity
- Memory affinity

The process affinity depends on how the PEs are mapped to physical cores. Meanwhile, the memory affinity depends on how the memory allocation is performed.

**Process Affinity**

The process affinity depends on how the pinning of PEs to physical cores is performed. When the pinning is performed the overall structure of the systems environment is determined. A problem arises where the pinning of PEs to physical cores is not profitable with the memory accessing pattern \[37\]. An example of a non-profitable pinning for an application of data exchange between neighboring PEs would be to pin the PEs in a round-robin fashion, the first process is pinned to a NUMA node and next is pinned to next NUMA node. This leads to that all PEs needs to transfer all data located in other PEs memory region over the interconnection network. When clearly there is no need for all processors to use the interconnection network with an optimal pinning. The impact of this problem can be lowered by rewriting the program according to the hardware configuration, but then the code would not be adaptable to any other hardware setup \[17\].

To solve this and to make software more adaptable to multiple hardware, some HPC systems and the Linux command `numactl` provide the possibility of doing your own pinning of PEs to cores from software \[38\][1]. Cray for instance, has an Application Level Placement Schedule that is called aprun. When launching an application with the aprun command there is a flag to determine how the pinning should be performed \[1][16\].

The aprun command is similar to any other application enabler, but with more control over the system. Aprun uses the flag `-cc` to do the pinning of PEs to cores. A simple usage of the aprun command can be seen in Code Snippet 3.1 where the flag `-n` specifies the amount of PEs and the flag `-cc` determines the pinning \[1\]. The result from this pinning can be seen in Fig. 3.2, where PE 1 and 3 ends up on CPU two and PE 0 and 2 ends up on CPU one.

**Code Snippet 3.1: Example on how to use Cray Application Level Placement Schedule to pin processing elements to physical cores.**

```
aprun -n 4 -cc 0,2,1,3 ./app.o
```
CHAPTER 3. MATERIALS AND METHODS

Figure 3.2: Code Snippet 3.1’s pinning of the processing element to cores. Both NUMA nodes have two cores and their own local memory area.

Memory Affinity and Access Policies

The process affinity depends on the way a PE is mapped to a core. Meanwhile, the memory affinity depends on where the memory is allocated by the kernel. How the kernel allocates memory is up to the specification of the kernel and the operating system.

Memory Policy

In a Linux environment, it is possible to both monitor and change the active memory policy of the current process, with regular system calls. To obtain the current memory policy of the system, the system call get_mempolicy is used [4]. In a standard Linux environment there are three possible memory policies MPOL_PREFERRED, MPOL_INTERLEAVED and MPOL_BIND [4]. In the MPOL_PREFERRED mode the kernel will try to allocate memory on the preferred memory node first and when the preferred memory node is out of memory the kernel continues to the next memory node. This mode can be used to create a local first memory allocation. To specify a custom allocation then the policy to use is MPOL_BIND. With the MPOL_BIND memory policy, the programmer can choose which memory nodes to use and which to not use. However, MPOL_BIND is not optimal on a NUMA architecture, due to the node is chosen in an ascending order. In some applications it could be profitable to only use a set amount of memory nodes, then MPOL_BIND could be an option [4]. For a UMA system, this could be optimal if the memory wants to be collected to a single node first then go on to the next one. When the last node specified by MPOL_BIND is out of memory there is no more free memory, even though other nodes could have memory left. The MPOL_INTERLEAVED mode allocates the first page on one node and the next one on another node. This is used in booting of computers.

First Touch Policy

When programming C and allocating dynamic memory it is most intuitive to think that the memory policy will be applied and then a memory area exists somewhere in the hardware. A simple example could be the following: when allocation 20 bytes of data with malloc then a 20 bytes region have been created and it is
3.2 Parallelization of Code

allocated somewhere in the hardware’s memory according to the current memory policy. In most cases of serial programming and distributed memory programming this intuitive picture is sufficient. But this is not the case. What is actually occurring is that the memory policy is invoked when the system needs to allocate memory for a virtual address. If a page in a process memory has not yet be modified it actually does not have any memory assigned to it. When an uninitialized memory position is modified for the first time the process will generate a hardware fault, also known as a page fault. During this page fault, the kernel of the CPU handles the allocation of the page with the current memory policy. Later when the page is allocated, the instruction that caused the page fault can resume and the program is carried one as intended [32] [39].

This means that the memory policy is applied when the memory is first used. Therefore, the name First Touch policy [29]. In serial programs, this is not really a problem because the memory will be allocated at some point with the current memory policy in the used process. But when handling a NUMA system and a shared memory model this could have a huge impact on execution time if not handled right.

If the memory policy is MPOL_PREFERRED the allocation of any data could end up in the state that one PE on a NUMA node owns the data virtually, but it is allocated on a different physical NUMA node because some other PE was the first one to write to the virtual page.

```plaintext
1 if(rank == 1)
2   memory[10] = 4712; // virtually processor 0
3 if(rank == 0)
4   memory[-10] = 10; // virtually processor 1
```

The simple Code Snippet above presents how PE 0 writes to an exposed virtual memory area which is owned by PE 1. As a result, this is an example where a PE could be the virtual owner of the page but not the physical owner of the memory. In most cases when using shared memory it is a good practice to initialize the memory before using it. Initializing the memory is performed by writing some dummy value to the memory before using it, to generate the page fault in the right PE.

### 3.2 Parallelization of Code

When going from a serial to a parallel program there is a need to do a decomposition of the data. For stencil based applications this is usually a quite easy task, due to the big amount of identical work that should be performed on each position in the grid. For example, the theoretical speedup is as high as when a PE updates a single position in the grid. More about the implementation in section 4.2. When parallelizing data intensive calculations it is of most importance how the data is partitioned between processes. For example, if the caches are not used to the fullest the applications might have lower performance than intended. When parallelizing grid-based application there are two easy ways of partition the data, either a row
CHAPTER 3. MATERIALS AND METHODS

Figure 3.3: The two ways of domain decomposition: the one-dimensional is presented on the left and the two-dimensional is presented on the right.

wise or block wise partition. When using the block-wise decomposition the data is divided into different blocks. The main concept which makes the block decomposition different from row decomposition is that a row decomposition is partitioned in one-dimension and the block decomposition is partitioned in two-dimensions [27]. In Fig. 3.3 both of these decompositions are partitioned for four PEs.

When using a similar amount of cores to the amount of data the block decomposition can have a substantial improvement over the row decomposition. But when the size of the grid is vastly larger than the number of processors, then the dimension of the decomposition does not have a huge impact on the execution time [22]. For this thesis, the partition of data will be partitioned by using the one-dimensional row decomposition.

3.3 MPI for Message Passing

The Message Passing Interface started as a standard to create a standardized way of using the message passing model. The message passing model builds on the property that all PEs have their own local address space and are able to share data by the concept of messages. Messages are communicated between a sender and a receiver over a "network". For this reason, the message passing model does not require a direct connection between PEs. With these properties, the message passing model matches the distributed memory model rather than the shared memory model. In the message passing model, communication involves both the sending and the receiving PE. The message is transferred from one point to another point, more commonly known as point-to-point communication [20]. Later in the report MPI message passing model will be addressed as MPI two-sided.

The two main calls for using a point-to-point communication are the sending of data MPI_Send and the receiving of data MPI_Recv. Both the functions MPI_Send and MPI_Recv are defined to be blocking. This means that MPI_Send wait for a confirmation by the receiving PE. Meanwhile the receiving PE is waiting until the sending message arrives and when the message has arrived, confirmation is sent back to sending PE before continuing execution. After the message has arrived and unpacked, the data is present in the receiving PE's local address space [20]. This can be seen in Fig. 3.4.
3.3. MPI FOR MESSAGE PASSING

Figure 3.4: This figure shows the different delays when using blocking communication. The blue bold part of the path is idle time.

With the simple MPI_Send and MPI_Recv functionality, it is possible to build collective functions such as MPI_Bcast, MPI_Gather and MPI_Scatter. Collective functions are used for easy communication with all PEs. MPI_Bcast send a set of data from one PE to all others, MPI_Gather collects a set of data from all the PEs to a single PE and MPI_Scatter splits the data into chunks and distributes the chunks to the other PEs. All of these functions are blocking and waits for all other PEs to finish before continuing [20]. To know which of all PEs to communicate with there is an indication system. By first creating a communication world where all PEs is part of. Furthermore, each of the PEs gets a distinct id to be able to address individual PEs. All message passing based functions are using this communication objects to keep track of all PEs which will be part of the communication. When utilizing modern computer architecture it takes more time to acquire data located in a different address space than your own. A single round of communication takes approximately 1 to 100 µs to perform. This is calculated with the LogP model for current high performance systems [14].

When using the blocking communication the PEs are actually idling and waiting until the communication has been finished. Furthermore, during this idle time PEs are not able to do anything while waiting, even if there are possible operations to be performed. When there is a possibility to overlap communication with computation it should be used. For these cases, there are non-blocking communications options in MPI_Isend and MPI_Irecv. To be able to determine when the message has arrived there are both test and wait functionality. MPI_Test is a non-blocking function. Meanwhile, the MPI_Wait blocks until the communication is finished. MPI does not demand that a blocking send is matched with a blocking receive. Therefore, combining blocking calls with non-blocking calls is possible [20] [19].
CHAPTER 3. MATERIALS AND METHODS

3.4 MPI for Shared Memory Programming

In this section, we present MPI support for shared memory programming. Firstly, the new remote memory properties are described, secondly, the usage of shared memory windows is described and lastly, a simple example program of MPI shared memory programming is provided.

3.4.1 MPI RMA Memory Models

To create a more flexible way of programming with the new standard of Remote Memory Access and to utilize different hardware setups MPI RMA has defined two distinct memory models, an increment from the initial single model in MPI-2. These models are:

- separated model
- unified model

Both of these models are showcased in Fig. 3.5. Each memory model has two types of abstract windows, a public window and a private window. When accessing a memory window with local CPU operations, also known as regular load and store operations, the operations are always addressed to the private window. Meanwhile, the put, get and accumulate types of operations target the public window. The difference between the models is substantial where the separated model does not demand any hardware support for creating memory coherency. Instead, all memory coherencies are handled by MPI software calls. For clarification, whenever the separated model or the unified model is used it can be checked by seeing if the public and the private window is logically the same, as described in Fig 3.5. If the
3.4. MPI FOR SHARED MEMORY PROGRAMMING

Figure 3.6: Two examples of synchronizations call. Active target synchronizations are present on the left. Passive target synchronizations are presented on the right.

windows are identical then the unified model is used. Otherwise, the separated model is used [19]. The unified model demands hardware-managed coherence. Without the involvement of extra synchronization, an update to the public window will at some point be propagated to the private window and vise versa for local updates to the private window [30] [26] [19].

Even though it might seem easy to use the unified model, it is not trivial. Firstly it is not known in what order or at what point in time the RMA operations have finished. It could have been in any order and in a non-deterministic way. To create an order of operations and an insurance of completion of memory calls to the window MPI provides window synchronizations. Secondly the bothersome property of not knowing when an update has been propagated from the public window to the private window [19].

To be able to reason about when an update has been propagated to the window there are synchronization operations and MPI concept of epochs are used. An epoch is defined by a set of synchronization operations. For example, MPI_Win_fence closes and re-opens an epoch [26]. Each one-sided communication on a window maps to a single epoch. Furthermore, epochs can be divided into two types, access and exposure epochs. If a PE is in an exposure epoch then the process-local window can be accessed by other PEs. In other words, the window is exposed to one-sided operations from other PEs. If a PE is in an access epoch then the PE can access remote memory. To be able to communicate from a PE to another, at least one of the PEs needs to be in an access epoch and the other needs to be in an exposure epoch. Fig 3.6 shows an example of the different epochs. When closing an epoch all of the RMA operations will be forced to completion of both locally and remotely [26].

In MPI there are two types of synchronizations. They are active target and passive target synchronization. When the active target synchronization is used, both of the PEs involved in the communication needs to be aware of the action. Aware in the sense that the target PE needs to open an exposure epoch to let the
other PEs access its window. In Fig 3.6 there are four different synchronization operations which are used in the active mode. To open and close an access epoch the function MPI_Win_start and MPI_Win_complete are used. To create an exposure epoch the functions MPI_Win_post and MPI_Win_wait are used.

When using passive target synchronizations the targeting PE is not actively involved in the communication. As a result, this creates a more true one-sided communication. This means that in the passive target synchronization model the exposure epoch is of no use. Hence, the memory window is always exposed to all other PEs. In Fig 3.6 it can be seen that some synchronization mechanisms are used. First, the calls to MPI_lock_all and MPI_unlock_all creates an access epoch. All PEs can freely read and update the memory of all other PEs in the communicator. To create an order MPI_Win_flush and MPI_Win_flush_local can be used. MPI_Win_flush ends and reopens an access epoch for each PE that has pending RMA operations [26]. In both models, it is possible to use the synchronization call MPI_Win_sync on the window object. This operation will not only synchronize the private and the public window of the calling window, it will also complete all pending memory calls to the window.

3.4.2 MPI Shared Memory Windows

To be able to use the shared memory properties of an MPI_Window is required that all PEs are connected to the same shared memory system. To have a software base mapping of the PE and to make the standard hardware independent, MPI decided to add the functionality of splitting the communicator on a property, instead of only distinct numbers. The function MPI_Comm_split_type is called with the split type MPI_COMM_TYPE_SHARED then the communicator is divided based on shared memory [19].

To create a window with the property of addressing other PEs memory with regular load and store operations the usage of MPI_Win_allocate_shared is essential. Like all other window creation commands, this is a collective call called by all PEs in the communicator, where each PE can define an individual size to allocate. If nothing else is stated the memory in the window is allocated continuously. Like other malloc like memory allocations a pointer is returned, which points to the start to the allocated memory area. The allocation window can be seen in Fig 3.7. Due to the memory area is contiguous the possibility to calculate the start position of other PEs memory windows exists. An in depth explanation can be seen in section 3.4.4. Even though the pointer of one PE has the address to the start position of their memory area this address should never be communicated between PEs. The different PEs could have different virtual memory mapping. As a result, the pointer could be pointing anywhere in the reviving PE’s memory. To receive the base pointer for another process memory area MPI have defined the function MPI_Win_shared_query. MPI_Win_shared_query is called with the window object and the other PE’s rank. The function returns not only the corresponding pointer but also the size of the window allocated by the same PE and the size of
3.4. MPI FOR SHARED MEMORY PROGRAMMING

Figure 3.7: Window contiguous allocation. The black parts are the starting allocation for a PE. PE 2 and PE 3 allocate different memory sizes.

With the property of setting the window to allocate 0 bytes, it could be hard to determine which PE is the owner of the starting position of window segment. For simplicity, MPI specifies an intuitive way of retrieving the start of the memory window. By calling `MPI_Win_shared_query` with MPI_PROC_NULL as rank, the start of the memory area will be returned [19]. The different possible outcome of the `MPI_Win_shared_query` can be seen in Fig. 3.8.

As stated before if the info object is not modified the memory of the window is allocated continuously. In systems with NUMA architecture allocating memory on a single NUMA node could lead to an imbalanced access time for different PEs placed on different NUMA nodes. It is of utmost importance to be able to allocate memory on the right NUMA node to achieve the lowest possible access time to memory. For the possibility of creating non-continues memory the function `MPI_Win_allocate_shared` has an info object. If the info object has the parameter `alloc_shared_noncontig` set to true then the allocation policy is now more loosely defined and each process can allocate memory according to their own memory policy.

An obvious impact of this is that it is no longer possible to calculate the offset to other PEs shared memory window. Therefore, the only way to acquire the other PEs is by using the `MPI_Win_shared_query` functionality. When using the shared memory for direct load and store operations no consistency or coherence is guaranteed by the MPI standard. Therefore, explicit calls to synchronize the

Figure 3.8: The "initial" pointer points to the start of the respective window. "PROCESS_NULL" is the pointer from `MPI_Win_share_query` with rank MPI_PROC_NULL. "Other" is the pointer from `MPI_Win_share_query` with the other processes rank.
CHAPTER 3. MATERIALS AND METHODS

memory could be needed. Although, the concept of using MPI_Win_sync is still of importance, due to the possibility of completing all pending memory operations targeting the window.

3.4.3 Allocation of Shared Memory

How is the non-contiguous memory implemented on real systems then? The standard only states that the memory should be non-contiguous and gives an advice to the implementer to use either padding or specific memory regions [36].

Padding

Allocate the data of that is called by the processor then move the next memory area to be allocated to the next page [25]. The advantages of this approach are that it is easy to use and the implementation only needs to have a pointer to the area and knowing all the different positions of the base pointers of the window. It can be seen as a long contiguous memory area with some holes in it.

Special Memory Segments

Implementing special memory segments is difficult because of the difference in the virtual addresses of the window. All the processes needs a distinct pointer to each of the starting positions of each part window. Because the differences in address translation, allocation of special memory segments could require more time than the padding version. Also it can lead to an increment of complexity in the implementation [39].

3.4.4 Address Calculation

In the contiguous model of the shared memory window, the virtual memory position returned by the MPI_Win_allocate_shared is a large contiguous segment. This leads to the possibility of calculation the memory addresses to any other PE if all allocation sizes are known. When wanting to address the start of any other PE’s window, with a lower rank than itself, it is only to take the interleaving PE’s memory sizes including the wanted PE’s size. When wanting a process which has a higher rank then add all interleaving memory sizes and your own memory size. Both of these statements are deduced from eq(3.1).

\[
\text{pointer}_{n+1} = \text{pointer}_n + \text{size}_n \Leftrightarrow \text{pointer}_{n-1} = \text{pointer}_n - \text{size}_{n-1}
\]

(3.1)

When having the start position of the memory area there is no problem to address the memory in the same way as if it would be any other memory area. As a result to C does not make any distinguish between pointers [19].
3.4. MPI FOR SHARED MEMORY PROGRAMMING

3.4.5 Code Example Using MPI Shared Memory

Using MPI shared memory windows can be performed in many different ways. The Code Snippet 3.2 explains how the window is created, used and later destroyed in an easy way.

Code Snippet 3.2: A simple program of shared memory programming with MPI shared memory windows.

```
1 function ()
2 MPI_Init(...)
3 MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, 0,
4   MPI_INFO_NULL, &shared);
5
6 MPI_Comm_rank(shared, &id);
7 size = id+1 * 10; // id starts at zero.
8 MPI_Win_allocate_shared(size*sizeof(int), sizeof(int), MPI_INFO_NULL,
9   shared, &memory, &window);
10
11 MPI_Win_lock_all(0,window);
12
13 if(rank == 1)
14   *(memory-10) = 4712; // knowing continues memory
15 if(rank == 0)
16   *(memory+10) = 10; // knowing continues memory
17
18 MPI_Win_sync(window);
19 MPI_Barrier(shared);
20
21 MPI_Win_unlock_all(window);
22
23 printf("rank = %d, memory = %d\n", id, *memory);
24
25 MPI_Win_free(&window);
26 MPI_Finalize();
```

When programming [MPI] the first call to do is the MPI_Init call. Hence, this starts an [MPI] section. MPI_init creates everything which is needed to run the MPI functionality. After starting an [MPI] section there is the need to definitely splitting the MPI_COMM_WORLD based on the shared memory property. The comm object, shared, has the property of creating shared memory windows. On line 8 the shared window is created with a call to MPI_Win_allocate_shared. Each [PE] allocates their own amount of memory in a contiguous block. [PE] 0 will allocate 10 integers and [PE] 1 will allocate 20 integers and so on. The pointer, memory is pointing to the start of the rank locally created memory area. After the memory is created the window enters an access epoch by each rank makes a call to MPI_Win_lock_all. The MPI_Win_lock_all function is not collective but often used collectively to assure that all ranks have an access epoch to all other ranks. The opening of an access epoch is to use the MPI_Win_sync on the window to forcing an update of the memory before reusing it again.
CHAPTER 3. MATERIALS AND METHODS

In the access epoch, PE 1 will write 4712 to PE 0’s base pointer and PE 0 will write 10 to PE 1’s base pointer. This is performed with a calculation of the memory distances between the two PE. The MPI_Barrier is to ensure that both of the ranks have written the value to the other rank before printing the value. No more is achieved in the epoch. Therefore, it is closed with the matching call to MPI_Win_unlock_all. The value of the base pointer is printed from each PE. The window is free with a call to MPI_Win_free and the MPI section is finished with a call to MPI_Finalize.
Chapter 4

Benchmark Applications

In this chapter, the two applications, matrix-matrix multiplication and Conway’s game of life, are presented. Also, how they are implemented with OpenMP, two-sided, and MPI shared (sections 4.1 and 4.2) is presented. The systems used for the experiment are described in section 4.3 and how the measurements are performed is presented in section 4.5.

4.1 Matrix-Matrix Multiplication

As defined by the benchmark the matrix-matrix multiplications should be evaluated for two dense matrices \( C = AB \). For the simplicity of the application implementation the algorithm chosen is the standard simple matrix-matrix multiplication, which has the time complexity of \( O(n^3) \). The serial code of the matrix-matrix multiplication can be seen in Code Snippet 4.1.


```c
\Transpose
  tmp[x][y];
  for (i = 0; i < x; ++i)
    for (j = 0; j < y; ++j)
      tmp[i][j] = B[j][i];

\Multiplication
  for (i = 0; i < x; ++i)
    for (j = 0; j < y; ++j){
      temp = 0;
      for (k = 0; k < y; ++k)
        temp += A[i][k] * tmp[j][k];
      C[i][j] = temp;
    }
```

To create data locality there is a transpose of the B matrix, which could improve the performance.
4.1.1 MPI Shared

In the MPI shared version, the memory of the application is allocated with the `MPI_win_allocate_shared`. A, B, C are therefore shared between all PEs. As can be seen in Code Snippet 4.2, the memory has been exposed to all PEs.

Code Snippet 4.2: Pseudocode of matrix-matrix multiplication with MPI shared.

```c
my_row = rows/number_of_processes;
MPI_Comm_split_type(..., shared);
...

MPI_Win_allocate_shared(col*my_row, ..., MPI_INFO_NULL, shared, &A, &a_w);
MPI_Win_allocate_shared(col*my_row, ..., MPI_INFO_NULL, shared, &B, &b_w);
MPI_Win_allocate_shared(col*my_row, ..., MPI_INFO_NULL, shared, &C, &c_w);

MPI_Win_shared_query(b_w, MPI_PROC_NULL, ..., ..., B);
MPI_Win_lock_all(0, a_w);
MPI_Win_lock_all(0, b_w);
if(id==0){
    initA();
    MPI_sync(a_w);
    initB();
    B=transpose(B);
    MPI_sync(b_w);
}

MPI_Win_unlock_all(a_w);
MPI_Win_unlock_all(b_w);
MPI_Barrier(shared)

\\\\\\\\
Multiplication
for (i = 0; i < my_row; ++i)
    for (j = 0; j < col; ++j){
        temp = 0;
        for (k = 0; k < col; ++k)
            temp += A[i][k] * B[j][k];
        C[i][j] = temp;
    }
```

The partition of memory is performed to have each of the PEs be the owner of an individual working part of the A matrix. This will lead to that each PE will only update the same part of the C matrix as they use in the A matrix. To be able to complete the matrix-matrix multiplication all points in the B matrix is required. The allocation of the B matrix will be divided so each process has the possibility of having some locality when using multiple NUMA domain. The synchronization of the window is only to promise that the memory is updated in each process before
4.1. MATRIX-MATRIX MULTIPLICATION

starting the actual matrix-matrix multiplication.

4.1.2 MPI Two-Sided

In the message passing model, PEj's memory is not addressable by other PEs directly. Hence, to keep the property of no redundant memory each processing element needs to only allocate a part of each matrix. For both the A and C matrix there is no problem, due to the fact that there is no need to accesses any other part than its own. However, all instances of the B is needed to complete one row of the matrix-matrix multiplication. As recalled the row based decomposition is used for the dividing of data. Therefore, a round based algorithm, to receive all the decomposed parts of the B matrix, is need. The algorithm is presented below.

1. Allocate part matrices and initialize them.
2. Perform calculation with current partB.
3. Send partB to next process recv partB from the previous process.
4. Repeat step 2 and 3 for n times.

This round based algorithm is known as the 1D systolic [41]. To clarify, the communication is performed in a ring scheduling. The lower rank is sending the data to the higher rank. The sending of data is repeated for n times to make sure that all ranks have all the parts of the B matrix. The communication part of the algorithm uses a MPI_Isend and MPI_Recv structure, see section 3.3 for further information. First using a non-blocking send and then using a blocking receive to wait for the message to arrive from the lower rank before continuing on the next round.

4.1.3 OpenMP

In the OpenMP version the plan is to create the same parallelism as with the MPI shared version, but using OpenMP pragmas. The outermost for loop will be parallelism with the for schedule(static) to obtain almost the same amount of computations in each of the threads. By using the private directives it is possible to prevent overwriting scope shared variables. The variables i, j, k and temp are privatized for each PE and a local copy of these variables is copied into their own address space. The pseudocode can be seen in Code Snippet 4.3.
CHAPTER 4. BENCHMARK APPLICATIONS

Code Snippet 4.3: Pseudocode of matrix-matrix multiplication code with OpenMP.

```c
A = malloc(col * row)
B = malloc(col * row)
C = malloc(col * row)
...
initA()
initB()
B = transpose(B)

#pragma omp parallel shared (A,B,C) private(i,j,k,temp)
{
#pragma omp for schedule (static)
for (i = 0; i < row; ++i)
  for (j = 0; j < col; ++j){
    temp = 0.0;
    for (k = 0; k < col; ++k)
      temp += A[i][k] * B[j][k];
    C[i][j] = temp;
  }
}
```

4.2 Conway’s Game of Life

In parallel computing, stencil calculations are a cornerstone of parallel algorithms. It is used to solve PDE, computational electromagnetic and SOBEL filters. Stencil computations are good for parallelization due to the big amount of data parallelism. Each single position in the data space should be evaluated with the same function. There are among others 4-point stencils and 8-point stencils on a two-dimensional grid. The 4-point and 8-point stencil is presented in Fig. 4.1. For higher dimensional grids different patterns could be used. Conway’s game of life is a simple game of cellular automaton. The game is turn-based game, played on a two-dimensional grid, where each cell can have one of two states, dead or alive. Each position in the grid is represented by a cell. In every turn of the game, each cell is evaluated by the rules of the game and updated to next turn.

![Figure 4.1](image)

Figure 4.1: The figure presents two different types of stencil operations. The red middle point is the current position and the surrounding blue points are the positions used for the stencil operation. A 4-point stencil is presented on the left and an 8-point stencil is presented on the right.
4.2. CONWAY’S GAME OF LIFE

The rules of the game are:

• Dead to Alive: If exactly three of the neighbors are alive then the cell will be alive in next turn.

• Alive to Dead: If the cell is alive and four or more of its neighbors are alive then the cell will be dead next turn.

• Alive to Dead: If the cell is alive and only one or less of its neighbors are alive then the cell will be dead next turn.

• Alive to Alive: If the cell is alive and two or three of its neighbors also are alive then the cell will be alive next turn. [13]

Conway’s game of life is a good example of an 8-point stencil application. In the game’s definition, there are an unlimited number of turns and the grid has an infinite size. With the transformation the game to an implementable computer program there arise some problems. In a computer program it is impossible to have an infinity grid size, hence of the restriction of not having infinite memory. With this restriction there are mainly two ways of transforming the game. Firstly the fixed transformation, which adopts the game in such a way that the edges of the grid never change [40], seen to the left in Fig. 4.2. Therefore, the initial data declared in the edges of the grid has a larger effect on the game in total. The advantages of this approach are that it is easily implemented and debugged. The disadvantages are that the impact of the edges is higher than the other cells in the grid and the changing grid is smaller \((\text{column} - 2) \times (\text{row} - 2)\) due to this approach. Secondly the tours transformation, that adopts the game in such a way that the edges are evaluated in a wrap-around fashion [40]. For example, the bottom edge takes the top edge as the input for the 8-point stencil, which can be seen to the right in Fig 4.2. The advantages of this approach are that every cell has the same amount of importance in the initial step and the grid is \(\text{row} \times \text{column}\) in the data. The disadvantages are that it is more special cases to implement and it is harder to evaluate correctness. Furthermore, when taking the memory layout into consideration this implementation could lead to an increased amount of cache misses if the data is large. When implementing a turn-based algorithm the turn itself usually is addressed as a computational cycle.

Figure 4.2: The Figure represents the two different approaches of transformation in an \(8 \times 8\) grid. The red area represents current position, blue represents the stencil and grey represents the non-evaluated positions. The gray area is only applicable in the fixed transformation on the left. The torus transformation is presented on the right.
The implementation of Conway’s game of life in a programming environment could be implemented in this approach:

1. Allocate two grids (new, old) of the size $column \times rows$.
2. Set the initial state of Conway’s game of life in both of the grid.
3. Use the stencil on all cells in the old grid and save the result to the new grid.
4. Swap grids.
5. Repeat step 3 and 4 until end of computational cycles.

The stencil application is presented in the Code Snippet 4.4 below. The 5th step of the application is a termination approach. This termination approach can be implemented after X amounts of computational cycles or after a special condition is met.

Code Snippet 4.4: Code for Conway’s game of life stencil operation.

```c
void stencil(int col, int row) {

    int nsum = old[col-1][row-1]+old[col][row-1]+old[col+1][row-1]
               + old[col-1][row ] +old[col+1][row ]
               + old[col-1][row+1]+old[col][row+1]+old[col+1][row+1];
    switch(nsum){
    case 3:
        new[i][j] = 1; break;
    case 2:
        new[i][j] = old[i][j]; break;
    default:
        new[i][j] = 0;
    }
}
```

The stencil code corresponds to the four rules of the Conway’s game of life. The first case of the stencil application implements the first rule and half of the fourth rule. The second part of the case corresponds to the second part of rule four, but also if the cell is dead, it should still be dead. The default case is that the cell is set to dead, even if it was dead before.
4.2. CONWAY’S GAME OF LIFE

4.2.1 MPI Shared

In this implementation of the parallel stencil application, the MPI shared memory is applied.

Code Snippet 4.5: Implementation of Conway’s game of life with MPI contiguously shared memory.

```c
... MPI_Comm_split_type(...,shared)
...
MPI_Win_allocate_shared(my_col*my_row*2, ..., MPI_INFO_NULL, shared, &
my_base, &window);

MPI_Win_shared_query(window, MPI_PROC_NULL, &size, &disp, &start_win);
MPI_Win_lock_all(0, window);
if(id == 0)
  InitMatrix(Old, start_win); InitMatrix(New, start_win+(col*row));
part_old = start_win+(my_col*my_row*id);
part_new = part_old+(col*row);
while(round-- >= 0) {
  for each cell {
    if(id == 0){
      skip_top_row
      stencil
    } else if(id == id.last) {
      skip_bottom_row
      stencil
    } else {
      stencil
    }
  }
  swap_pointers();
  MPI_Win_sync(window);
  MPI_Barrier(sharde);
}
MPI_Win_unlock_all(window);
...
```

In this implementation, the grid is allocated in a sequential way, meaning that the old grid is allocated before the new grid, as can be seen in Fig. 4.3. When defining the pointers in such a way as can be seen in Fig. 4.3 could lead bad memory locality. Because, when each PE is working on both halves of the memory area, then the data needs to be transported over the interconnection network in a NUMA system. In Code Snippet 4.5 there is a skipping of the top and the bottom row in the algorithm. This is only to not have the edges updated, due to the using the fixed transformation.
4.2.2 MPI Shared Non-Contiguous

In this implementation, the non-contiguous memory allocation is used. The difference between the continuous implementation and the non-contiguous implementation is that there is no possibility of calculating the address to other processes memory in the non-contiguous model. Instead, to acquire the process bordering memory areas, the `MPI_Win_shared_query` is used with the higher and lower ranks. The difference in how to allocate of the memory can be seen Code Snippet 4.6.

Code Snippet 4.6: Implementation of Conway’s game of life with MPI shared non-contiguously memory.

```c
MPI_Win_shared_query(window, id-1, &size, &disp, &start_lower);
MPI_Win_shared_query(window, id+1, &size, &disp, &start_higher);
old_lower = start_lower + ((my_row-1) * (my_col));
new_lower = old_lower + (my_row) * (my_col);
old_high = start_high;
new_high = old_high + (my_row) * (my_col);
...
```

Instead, the usage will now be to obtain the pointers from the PE’s that has the window position lower and higher than current PE, then allocate the virtual memory for both grids in the same allocation. The virtual memory is mapped to have half
4.2. CONWAY’S GAME OF LIFE

Figure 4.4: Allocation of the MPI non-contiguous shared memory for Conway’s game of life. The yellow part of the memory allocation by PE 0 and the blue part is allocated by PE 1.

of the memory corresponding to the old grid and the other half corresponding the new grid. The virtual mapping of the cores are now correct in the sense of putting the working data in the right virtual area see Fig. 4.4.

4.2.3 MPI Two-Sided

Parallelizing Conway’s game of life with MPI two-sided has some drawback. When dividing up the data between the processes, each PE will obtain a fixed area of the grid to update. But for the boundaries of the data area, the first and last rows of data, the data to perform the stencil operation is not present. A simple solution is to allocate an extra row of data at each border and has the owner of the data to transmit it to the needing PE. These extra data points are called ghost cells [11], because the PE is not the owner of the data, but still needs them for the calculations of its data points. In Fig 4.5 the ghost cell is presented on an $8 \times 8$ matrix with two PEs.

A non-blocking send request `MPI_Isend` and a blocking receive `MPI_Recv` is used. This is to use one of the most important of the message passing system, to overlap communication with computation. Otherwise, the system would be idling when waiting for a response that the message has been delivered before continuing, see section 3.3 for more information. In the Code Snip 4.7 this is represented by the `send_ghost` and `recv_ghost`. The computation between the communications is the stencil operation on the data which is not using the ghost areas data. This is represented as `stencil_my` in the Code Snippet 4.7.
Figure 4.5: Ghost cells for a $8 \times 8$ matrix with two PEs. Each PE has a ghost row which is the other PE’s boundary row. The orange part is the ghost cell from PE 0 to PE 1 and the purple is from PE 1 to PE 0.

Code Snippet 4.7: Implementation of Conway’s game of life with MPI two-sided by using ghost cells.

```c
...
part_old = malloc(my_col * my_row * sizeof(int));
part_new = malloc(my_col * my_row * sizeof(int));
ghost_low = malloc(1 * my_row * sizeof(int));
ghost_high = malloc(1 * my_row * sizeof(int));
...
if(id == 0)
  InitMatrix_and_send(Old, start_win);
...
while(round-- >= 0) {
  send_ghost
  if(id == 0){
    stencil_my
    recv_ghost
    stencil_bottom
  } else if(id == id.last) {
    stencil_my
    recv_ghost
    stencil_top
  } else {
    stencil_my
    recv_ghost
    stencil_bottom
    stencil_top
  }
  MPI_Barrier(MPI_COMM_WORLD);
  swap_pointers();
}
...
```

In the function `init_and_send` the file is read and partitioned out to the other PEs by the master rank.
4.3 Benchmark Environment

The system used for the measurements and the evaluation are:

- Cray XC40 system, Beskow, using a single node.
- Haswell node, Tegner, using a single node.

4.3.1 Beskow

Beskow is an XC40 system consisting of 53632 cores, located at the KTH Royal Institute of Technology, Stockholm. At its peak time, it was placed 32 on TOP500’s list of world fastest HPC.

Hardware

The system is built out of Intel Xeon Processor E5-2698 v3 Haswell chips. Each chip is operated on a die. The E5-2698 v3 chip has 16 cores and a 40 MB last level cache, implemented with Intel smart cache \[3\]. Two of these Haswell chips are mounted together to create a single XC40 computation node. The Haswell chip has 32 GB of memory, which leads to an XC40 node on Beskow has 64 GB of memory in total. To connect the two dies together there are two QPIs connected to transfer data between the two dies. Each of the QPI interconnects has a transfer rate of 9.6 GT/s. The 8 DDR4 SDRAM has a transfer rate of 2.13-4.26 GT/s each \[3\]. Fig. 4.6 provides a simplified image of the computation nodes architecture. In total each NUMA node has 16 cores and the XC40 computation node has 32 cores.

Software

Compilation performed on Beskow is carried out with the macro ‘cc’, which invokes the current compiler set to the module environment. To change the environment it is only to swap the module with the command module swap PrgEng. The compilers used are GCC 4.9.1, cray 8.3.4 and Intel 15.0.1. When compiling with any of these compilers the ‘-O2’ optimization flag will be used. The MPI compilation is also handled with the ‘cc’ compiler command and no other command will be used.
For OpenMP program, the OpenMP flag needs to be set. The MPI version used are cray-mpich 7.0.4.

4.3.2 Tegner

Tegner is a cluster system, at KTH Royal Institute of Technology, Stockholm. The system has four different types of nodes, two Haswell types and two Ivy bridge types exist on the cluster.

Hardware

The node which is used is built out of Intel Xeon Processor E5-2690 v3 Haswell chips. Each chip is operated on a die. Instead of Beskow’s 32 processes per node Tegner only has 24. But the memory placed on a Tegner node is larger, with 512 GB per node. Tegner has two processing chips, which have been mounted together. The hardware setup for intra-node communication is the same as for the Beskow system.

Software

Compilation on Tegner is performed with the regular compiler command. Therefore, it depends on what compilers which are loaded into the environment. All of the
4.4. SIMULATION SET-UP

compilation on Tegner uses the compiler flag `-O2`. For MPI code, the ‘mpicc’ (GCC) or ‘mpiicc’ (intel) command is run for the tests. For OpenMP program, the OpenMP flag needs to be set. The MPI version used are openmpi 5.1 and intelmpi 5.0.3.

4.4 Simulation Set-Up

In this section, the dataset used for verification and performance measurements is presented.

4.4.1 Matrix-Matrix Multiplication

When doing measuring of the matrix-matrix multiplication the datasets mmult_big, mmult_small and mmult_mini are used. Each of these matrices will be used as both A and B, but will be read into the memory in the same way as if it would be two different matrices. When evaluating strong scaling the dataset mmult_big will be used. mmult_big is a square matrix with $5120 \times 5120$ positions. The datasets mmult_small and mmult_mini matrices are only used for verification of the correctness of the implementations. The mmult_small matrix corresponds of $32 \times 32$ positions. Meanwhile, the mmult_mini matrix only corresponds of $3 \times 3$ positions, for illustrations purposes. Table 4.1 gives a summary of the datasets used for the matrix-matrix multiplication.

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>mmult_big</td>
<td>$5120 \times 5120$</td>
</tr>
<tr>
<td>mmult_small</td>
<td>$32 \times 32$</td>
</tr>
<tr>
<td>mmult_mini</td>
<td>$3 \times 3$</td>
</tr>
</tbody>
</table>

Table 4.1: Table of the different datasets and their sizes, which is used for evaluating matrix-matrix multiplication.

4.4.2 Conway’s Game of Life

When doing measurements of the Conway’s game of life the datasets strong_B, weak_B size-small, size-mini and Strong_T are used. For strong_B is corresponding of a grid of $5120 \times 5120$ cells to evaluate the strong scaling of the implementations on Beskow. For weak scaling the datasets used is $1024 \times 1024$, $2048 \times 1024$, $4096 \times 1024$, $8192 \times 1024$, $16384 \times 1024$, $32768 \times 1024$ and is referred to as weak_B. An additional test of $16 \times 16$ cells referred to as size-small and size-mini is used for showing the correctness between the serial version and the implemented versions of the Conway’s game of life. For Tegner the problem size for the strong scaling test is $5112 \times 5120$ cells and called Strong_T. In Table 4.2 the datasets used for Conway’s game of life are shown.
CHAPTER 4. BENCHMARK APPLICATIONS

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>weak_B</td>
<td>1024 to 32768 × 1024</td>
</tr>
<tr>
<td>strong_B</td>
<td>5120 × 5120</td>
</tr>
<tr>
<td>size-small</td>
<td>16 × 16</td>
</tr>
<tr>
<td>size-mini</td>
<td>8 × 8</td>
</tr>
<tr>
<td>strong_T</td>
<td>5112 × 5120</td>
</tr>
</tbody>
</table>

Table 4.2: Table of the different datasets and their sizes, which is used for evaluating the Conway’s game of life.

4.5 Performance Measurements

To perform timing test for parallel programs it is better to use the library’s own implementation of measuring time. When using MPI it is common to use the function MPI_Wtime. When using OpenMP the function omp_get_wtime is used. If the time is returned from more than one PE, then the worst case time is used.

Measurements for speedup and efficiency graphs are used to acquire an indication of how well the implementation is performing in regard to some reference implementation. In this thesis, all the reference implementations will be the serial implementation of the application. To collect viable measurements, each test is run for 15 times and then a mean as well as the standard deviation is taken. The real speedup is calculated with eq(4.1). The numerator part of the equation is the reference time from the serial version and the denominator is the parallel time for the same programs size with p PEs [15].

\[ S(p) = \frac{T(n, 1)}{T(n, p)} \] (4.1)

Efficiency in strong scaling eq(4.2) is calculated using the speedup with regard to how many PEs that was used. For a perfect scaling, the speedup should be the same as the number of PEs used [15].

\[ E_{\text{strong}}(p) = \frac{T(n, 1)}{pT(n, p)} = \frac{S(p)}{p} \] (4.2)

A perfectly scaling parallel program is having the same property as the Ideal Linear Bound. This is when all work is parallelized and the scaling factor is 100%. This means that the speedup is the same as the amount of cores running the program. But in most cases, a program has parts which cannot be parallelized. For those programs, it is theoretically impossible to reach the same performance as the Ideal Linear Bound. Therefore, instead of using the Ideal Linear Bound another normal bounder to use is the Serial Overheads Bound [23]. This is often represented by Amdahl’s law. Amdahl’s law can be seen in eq(4.3) where \( d \) is the percentages of the program that can be executed in parallel. One of
4.5. PERFORMANCE MEASUREMENTS

The preferences for eq (4.3) to be true is that the data evaluated (n) stays constant when increasing the numbers of PEs.

\[ S_{Amdahl}(p) = \frac{1}{1 - d + \frac{d}{p}} \]  

(4.3)

In this thesis, we will be using a serial file reading technique. Reading from the file and placing the data in memory will be considered as the serial part of the program.

Efficiency in weak scaling, each PE is handling the same amount of data even as the PEs is increased. This is to see how good the efficiency is, even though the memory does not fit on a single processor. Eq (4.4) shows the formula for calculation the weak efficiency [15].

\[ E_{weak}(p) = \frac{T(n,1)}{T(np,p)} \]  

(4.4)

4.5.1 Code Profilers

In parallel programs, it is common to use profilers to obtain extended knowledge of what is actually happening during a run of the program. Profilers can describe anything from how long time the run took to how many times a specific hardware event was triggered. For profiler, the tool CrayPat was used. CrayPat is a powerful performance analyzing tool and is built of three major components pat_build, pat_report and Apprentice2. Pat_build is used to create an analyzable program from a compiled object file. Pat_report creates a standalone text report that can be used to further explore the data generated, which has been produced from running the analyzable program. Apprentice2 is a graphical analysis tool that provides additional visualization possibilities than pat_report [6]. A simple example of using CrayPat can be seen in Code Snippet 4.8.

Code Snippet 4.8: The work flow how to use the profiler CrayPat on a cray systems.

```
cc -o profiling stencil.c  
//returns profiling+pat
pat_build profiling -g <option>
//creates a profiling+pat+<id>+<pid>.xf
aprun -n 1 ./profiling+pat
//creates profiling+pat+1245+1602t.ap2
pat_report profiling+pat+1245+1602t.xf
//Apprenence2
app2 profiling+pat+1245+1602t.ap2
```

By default the Apprentice2 measures among others Load Imbalance, function and region profiles and communication between different PEs. To modify what or
CHAPTER 4. BENCHMARK APPLICATIONS

how the pat_build does the measurement, pat_build is built to uses environment variables. The most important and possibly the easiest variable to use could be PAT_RT_SUMMARY. This variable is set as a default to 1, this means that the data produced is only a summary of all the data that could be interpreted. To change what performance counter events to be monitored the value of PAT_RT_PERFCTR can be changed. It can be changed to a predefined group of counters or a comma separated list. When doing the performance analysis, the environment variables will be set to PAT_RT_SUMMARY=0 and PAT_RT_PERFCTR=10 to measure all data possible and to check the caches data. To change from a sampling test to a trace gathering the call to pat_build needs a flag to be set. For this experiment the flag -g will be used with options to monitor either MPI or OpenMP [6].

CrayPat is also used to measuring the memory consumption. The measurement which will be used is the High Memory Water Mark. CrayPat uses the information from the file /proc/self/numa_maps at the end of the program to evaluate memory consumption. In the High Memory Water Mark the total size of all pages that were actually mapped into physical memory is used [6].

4.6 Verification Tests

A simple verification test is created to verify the correctness of the implementations. Correctness is evaluated with the help of the input-output behavior of the implementation. If the output produced by a parallel implementation is the same as the serial output, then the parallel implementation is correct. For simplicity, the output is checked by the Linux command diff, which compares the content of two files line by line [39]. How the command is used is described in Code Snippet 4.9.

Code Snippet 4.9: Example of how to use the diff command.

diff <correct> <output_to_validate>

4.6.1 Matrix-Matrix Multiplication

In the case of the verification test of the matrix-matrix multiplication, the code should provide the correct output. The correct implementation result is shown in Fig. 4.7 for the test dataset mmult_mini is used.

\[
AB = C \iff \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} = \begin{bmatrix} 30 & 36 & 42 \\ 66 & 81 & 96 \\ 102 & 126 & 150 \end{bmatrix}
\]

Figure 4.7: Matrix multiplication. The matrices A and B are represented as mmult_mini and use as input matrices. The C matrix is the output matrix.
4.6. VERIFICATION TESTS

4.6.2 Conway’s Game of Life

In our implementation of Conway’s game of life, we used the stop condition of \( X \) amounts of rounds, for more information see section 4.2. For the correctness evaluation, the amounts of rounds were chosen to be 1201. In Fig. 4.8 can the initial state of the grid be seen to the left. To the right of Fig. 4.8 can the grid be observed after 1201 rounds.

Figure 4.8: Snapshot of the initial state (left) and after 1201 rounds (right) of Conway’s game of life with the \texttt{stencil\_mini} dataset.
Chapter 5

Results

In this chapter, the results from the Benchmark Applications are presented. The result of the verification test is provided in section 5.1. Performance result with MPI shared is provided in section 5.2. Compiler difference between memory areas is presented in section 5.3 and lastly, performance and memory consumption are evaluated between MPI shared, MPI two-sided and OpenMP in section 5.4.

5.1 Verification Results

In this section, the correctness test from section 4.6 for matrix-matrix multiplication and Conway’s game of life is presented.

5.1.1 Matrix-Matrix Multiplication

For matrix-matrix multiplication, the verification tests were performed on both the datasets mmult_small and mmult_mini. Mmult_small was evaluated with 1, 2, 4, 8, 16, 32 PEs, meanwhile for the dataset of mmult_mini were only evaluated with 1 and 3 PEs. In Fig. 5.1 the outputs from the serial implementation and the MPI shared implementation can be seen.

```
30  36  42
66  81  96
102 126 150
```

Figure 5.1: The output from the mmult_mini test set when evaluated with different implementations of the matrix-matrix multiplication. The output of the serial version is presented on the left and the output of the MPI shared memory implementation is presented on the right.
5.1.2 Conway’s Game of Life

For Conway’s game of life, the verification tests were performed on both sets \textit{size-mini} and \textit{size-mini}. \textit{Size-mini} were evaluated with 1, 2, 4, 8, 16, 32 PE, meanwhile for the \textit{size-mini} tested for only 1, 2 and 4 PE. In Fig. 5.2 the outputs from the serial and the MPI shared implementations.

\begin{verbatim}
1 0 0 0 0 1 0 1
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1
1 0 0 0 0 0 1 1
0 0 0 0 0 0 1 0
1 0 0 0 1 0 0 1
0 1 0 1 0 1 0 1
0 1 1 1 0 1 0 0
\end{verbatim}

\begin{verbatim}
1 0 0 0 0 1 0 1
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1
1 0 0 0 0 0 1 1
0 0 0 0 0 0 1 0
1 0 0 0 1 0 0 1
0 1 0 1 0 1 0 1
0 1 1 1 0 1 0 0
\end{verbatim}

Figure 5.2: Output from the MPI shared implementation of Conway’s game of life is represented on the right side. The left grid is the output from the serial program, which is the control instance.
In this section, the performance of the two applications with the MPI shared implementations will be evaluated. In subsection 5.2.2 the effect of the NUMA nodes and affinity options are presented.

### 5.2.1 Matrix-Matrix Multiplication

![Graph showing speedup and efficiency for matrix-matrix multiplication with MPI shared windows on Beskow.](image)

Figure 5.3: Speedup measurement (left) and strong scaling (right) of matrix-matrix multiplication with MPI shared windows on Beskow.

In Fig. 5.3, the speedup and strong scaling of the matrix-matrix multiplication implementation provided in subsection 4.1.1 are presented. As can be seen from Fig. 5.3, the efficiency remains above 35% and is stabilizing, if utilizing all possible cores on the Haswell node in the Beskow system. In Table 5.1, the meantime, variance, speedup and efficiency is presented for 1, 2, 4, 8, 16, 32 cores using the dataset `strong_b`. The maximal speedup is 12.1x and the variance is overall very low.
CHAPTER 5. RESULTS

<table>
<thead>
<tr>
<th>Number of Processes</th>
<th>Speedup</th>
<th>Strong Scaling</th>
<th>Mean ($\mu$)</th>
<th>Standard deviation ($\sigma$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>100%</td>
<td>49.1s</td>
<td>0.31s</td>
</tr>
<tr>
<td>2</td>
<td>1.8</td>
<td>91.9%</td>
<td>26.7s</td>
<td>0.13s</td>
</tr>
<tr>
<td>4</td>
<td>3.1</td>
<td>77.5%</td>
<td>15.8s</td>
<td>0.04s</td>
</tr>
<tr>
<td>8</td>
<td>4.9</td>
<td>61.9%</td>
<td>9.93s</td>
<td>0.08s</td>
</tr>
<tr>
<td>16</td>
<td>7.8</td>
<td>48.9%</td>
<td>6.3s</td>
<td>0.07s</td>
</tr>
<tr>
<td>32</td>
<td>12.1</td>
<td>37.5%</td>
<td>4.08s</td>
<td>0.00s</td>
</tr>
</tbody>
</table>

Table 5.1: The mean time, variance, speedup and efficiency for the MPI shared implementation of matrix-matrix multiplication executed on Beskow. "Mmult_big" was used as dataset.

5.2.2 Conway’s Game of Life

In the implementation of contiguous memory the non-optimal placement of memory, as described in section 4.2.1, can be seen in Fig. 5.4. However, there is no noticeable difference in performance up to 16 Processing Elements. Therefore, no difference in performance can be seen between contiguous and no contiguous memory allocation. Provided by Fig. 5.4 the difference between placing half the working data in each NUMA node is 10%. Hence, the implementation 4.2.2 against 4.2.1.

![Figure 5.4: Speedup measurement (left) and strong scaling measurement (right) for the two MPI shared memory implementations of Conway’s game of life with initialization. "MPI Shared-No-Contiguous" is the implementation using no contiguous memory. "MPI Shared-Contiguous" is the contiguous implementation.](image-url)
5.2. PERFORMANCE RESULTS

In the experiment of Conway’s game of life, the impact of initializing the data is evaluated, on Beskow. In Fig. 5.5 the impact of not initializing the memory before using it is presented. This leads to the thought that MPI shared memory region is using the first touch policy, presented in section 3.1.2. When increasing the core count to 32 and utilizing both NUMA nodes it can be seen that the placement of the memory has a big impact on the performance. The increment of speedup between 16 and 32 cores is only around 10% for the non-initialized implementation. Meanwhile, when using the implementation with memory initialization that increment is around 64%.
In Fig. 5.6 the impact of process affinity is presented. Both configurations produce an almost equal amount of performance and there is not a huge difference in using any of these pinning properties for this application. The CPU configuration is represented as placing the amount of cores in an ascending order. Meanwhile, the Half configuration places the half of the cores in the first NUMA node and the last half in the other NUMA node.

<table>
<thead>
<tr>
<th>Number of Processes</th>
<th>Speedup</th>
<th>Strong Scaling</th>
<th>Mean ($\mu$)</th>
<th>Standard deviation ($\sigma$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.82</td>
<td>82.7%</td>
<td>50.5s</td>
<td>0.54s</td>
</tr>
<tr>
<td>2</td>
<td>1.6</td>
<td>81.4%</td>
<td>25.7s</td>
<td>0.33s</td>
</tr>
<tr>
<td>4</td>
<td>3.1</td>
<td>78.4%</td>
<td>13.3s</td>
<td>0.04s</td>
</tr>
<tr>
<td>8</td>
<td>5.7</td>
<td>71.6%</td>
<td>7.3s</td>
<td>0.02s</td>
</tr>
<tr>
<td>16</td>
<td>10.8</td>
<td>63.0%</td>
<td>4.1s</td>
<td>0.01s</td>
</tr>
<tr>
<td>32</td>
<td>17.5</td>
<td>54.7%</td>
<td>2.35s</td>
<td>0.01s</td>
</tr>
</tbody>
</table>

Table 5.2: The mean time, variance, speedup and efficiency for the MPI shared implementation of Conway’s game of life executed on Beskow. \textit{Strong\_b} was used as dataset.

In Table 5.2 the times of the fastest MPI shared Memory implementation is presented. The maximum speedup for Conway’s game of life MPI shared had 17.5x and 54.7% efficiency.
5.3 Comparison of Compilers Interaction with MPI Shared

When using applications it is important that the compiler can make a good translation from the high level code to the runnable machine code. To evaluate if there is any difference between malloc and MPI_Win_allocate_shared, a single computational cycle of Conway’s game of life is timed. Compilers which were evaluated were Cray, Intel and GCC. All tests are also using the cray-mpich implementation of MPI and are run on single Haswell node, using the Beskow system.

<table>
<thead>
<tr>
<th>Allocator</th>
<th>Cray</th>
<th>Intel</th>
<th>GCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_{\text{malloc}}$ (2 PE)</td>
<td>0.034 ± 2e−5</td>
<td>0.037 ± 3e−5</td>
<td>0.039 ± 6e−5</td>
</tr>
<tr>
<td>$t_{\text{MPI_shared}}$ (2 PE)</td>
<td>0.041 ± 2e−5</td>
<td>0.038 ± 2e−5</td>
<td>0.039 ± 3e−5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Allocator</th>
<th>Cray</th>
<th>Intel</th>
<th>GCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_{\text{malloc}}$ (4 PE)</td>
<td>0.017 ± 1e−5</td>
<td>0.019 ± 1e−5</td>
<td>0.020 ± 4e−5</td>
</tr>
<tr>
<td>$t_{\text{MPI_shared}}$ (4 PE)</td>
<td>0.021 ± 1e−5</td>
<td>0.019 ± 2e−5</td>
<td>0.019 ± 2e−5</td>
</tr>
</tbody>
</table>

Table 5.3: The time (in seconds) it takes to complete one round of the Conway’s game of life with different compilers when using malloc or MPI shared memory.

It is obvious from inspecting Table 5.3 that the fastest computational cycle is produced by the malloc pointer in the cray compiler. Meanwhile, the slowest pointer is produced by the MPI_Win_allocate_shared in the cray compiler. For the GCC and Intel compilers, there is no major difference in execution time between the two compilers.

Figure 5.7: The speedup measurements (left) and strong efficiency measurements (right) for MPI two-sided and MPI shared memory, when compiled with GCC compiler on Beskow.
Figure 5.8: Compiler performance difference between Intel and GCC compilers when using MPI shared on Tegner system. Speedup on the left side and strong scaling on the right side.

It is clear from Fig. 5.7 that the optimizations on the pointers are similar with the GCC and the Intel compilers. Here curves produce the same measurements for both 16 and 32 PEs. In Fig. 5.8 the performance difference between the Intel and GCC compiler is presented. This measurement were performed on the Tegner system. Hence, the amount of PEs only goes to 24. The Intel compiler has a slightly higher performance than the GCC compiler overall.
5.4 Performance Comparison of MPI Shared, MPI Two-Sided and OpenMP Implementations

In this section, the result from MPI shared, MPI two-sided and OpenMP implementations is presented for both matrix-matrix multiplication and Conway’s game of life.

5.4.1 Matrix-Matrix Multiplication

In this section, the three different implementations of parallelism will be evaluated, OpenMP, MPI shared and MPI two-sided.

![Graph showing speedup and efficiency for different implementations.]

Figure 5.9: Speedup measurements on the left side and strong efficiency on the right side for the implementations of MPI two-sided, MPI shared and OpenMP with the application matrix-matrix multiplication on Beskow.

In Fig. 5.9 (left) the speedup for 1, 2, 4, 8, 16, 32 PEs is presented and strong scaling is presented in Fig. 5.9 (right). Both OpenMP and MPI shared provide a reasonable speedup and efficiency curves. On the other hand, MPI two-sided has a dip in performance for 16 PEs, there was no increase in speedup. The OpenMP implementation has almost the same curves as MPI shared. In Table 5.4 the meantime, variance, speedup and strong efficiency is provided for 8, 32 PEs. In Fig. 5.9 it can be seen that OpenMP has a maximum difference between meantime is max 3%. Meanwhile, at 32 processes MPI two-sided and MPI shared has a difference in mean time of 10%, because of the NUMA architecture.
Table 5.4: The mean time, variance, speedup and efficiency for MPI two-sided, MPI shared and OpenMP implementations of matrix-matrix multiplication. *Mmult_big* was used as the dataset and the tests were run on Beskow.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Speedup</th>
<th>Speedup efficiency</th>
<th>Mean (µ)</th>
<th>Standard deviation (σ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI Two-Sided (PE 8)</td>
<td>4.9</td>
<td>61.7%</td>
<td>9.96s</td>
<td>0.01s</td>
</tr>
<tr>
<td>MPI Two-Sided (PE 32)</td>
<td>15.3</td>
<td>47.1%</td>
<td>3.21s</td>
<td>0.03s</td>
</tr>
<tr>
<td>MPI Shared (PE 8)</td>
<td>4.9</td>
<td>61.9%</td>
<td>9.93s</td>
<td>0.09s</td>
</tr>
<tr>
<td>MPI Shared (PE 32)</td>
<td>12.0</td>
<td>37.6%</td>
<td>4.09s</td>
<td>0.00s</td>
</tr>
<tr>
<td>OpenMP (PE 8)</td>
<td>5.0</td>
<td>63.2%</td>
<td>9.73s</td>
<td>0.02s</td>
</tr>
<tr>
<td>OpenMP (PE 32)</td>
<td>12.4</td>
<td>38.9%</td>
<td>3.94s</td>
<td>0.01s</td>
</tr>
</tbody>
</table>

Memory Usage

Memory consumption could be evaluated by the profiling data. In Table 5.5, a summary of the memory consumption per process is provided. Even tough, when using OpenMP, *CrayPat* only provides the total amount of memory, therefore all OpenMP data points are divided by the number of PEs used. The memory consumption of the MPI two-sided is higher than the usage of both MPI shared and OpenMP.

<table>
<thead>
<tr>
<th>PE 1</th>
<th>PE 2</th>
<th>PE 4</th>
<th>PE 8</th>
<th>PE 16</th>
<th>PE 32</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMP</td>
<td>421 MB</td>
<td>218 MB</td>
<td>117 MB</td>
<td>66 MB</td>
<td>41 MB</td>
</tr>
<tr>
<td>MPI Shared</td>
<td>72 MB</td>
<td>48 MB</td>
<td>35 MB</td>
<td>29 MB</td>
<td>26 MB</td>
</tr>
<tr>
<td>MPI Two-Sided</td>
<td>472 MB</td>
<td>276 MB</td>
<td>160 MB</td>
<td>98 MB</td>
<td>64 MB</td>
</tr>
</tbody>
</table>

Table 5.5: The memory consumption of the matrix-matrix multiplication application per process element, provided by CrayPat.

It can be seen from Table 5.5 and Fig. 5.10 that the memory consumption of the shared memory model (MPI shared and OpenMP) is lower than the message passing model (MPI two-sided). When measuring for 32 PE OpenMP with its threaded implementation has better memory consumption than MPI shared. Also, it can be seen that the memory consumption goes down for OpenMP for 16 to 32, this could be because of allocation different sizes with malloc depending on the amount of thread used. The default memory usage of both MPI two-sided and OpenMP is higher than MPI shared. But when using 32 PEs the OpenMP implementation has lower memory consumption than MPI shared.
5.4. PERFORMANCE COMPARISON OF MPI SHARED, MPI TWO-SIDED AND OPENMP IMPLEMENTATIONS

![Graph showing memory usage vs number of processes]

Figure 5.10: Total memory usage summed over all processing element. CrayPat’s High Memory Water Mark has been used for the measurement.

### 5.4.2 Conway’s Game of Life

Fig. 5.11 shows the performance of different implementations of the Conway’s game of life, the dataset used is *Strong_B*. In Fig. 5.11 the MPI shared implementation is strictly worse than the MPI two-sided implementation. Table 5.6 shows differences in the speedup for 4, 8, 16 and 32 PEs, the non-initialized MPI shared implementation is also presented, to be able to give OpenMP a different lower bound. The non-initialized MPI shared implementation is outperforming the OpenMP version.

<table>
<thead>
<tr>
<th></th>
<th>MPI Two-Sided</th>
<th>MPI Shared</th>
<th>MPI Shared no init</th>
<th>OpenMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 PE</td>
<td>3.8</td>
<td>3.1</td>
<td>3.1</td>
<td>2.6</td>
</tr>
<tr>
<td>8 PE</td>
<td>7.0</td>
<td>5.7</td>
<td>5.7</td>
<td>4.6</td>
</tr>
<tr>
<td>16 PE</td>
<td>10.9</td>
<td>10.6</td>
<td>9.9</td>
<td>6.2</td>
</tr>
<tr>
<td>32 PE</td>
<td>19.1</td>
<td>17.6</td>
<td>10.7</td>
<td>10.5</td>
</tr>
</tbody>
</table>

Table 5.6: The speedup of MPI two-sided, MPI shared, OpenMP, and MPI shared without initializing the memory, for Conway’s game of life.

Table 5.7 shows the speedup, scaling, mean and variance for 8 respectively 32 PEs. The OpenMP version has a big variance for 8 PEs. This could be due to awaking and sleeping thread and pinning of OpenMP memory addresses. The
Figure 5.11: Speedup measurement (left) and strong scaling measurement (right) for MPI two-sided, MPI shared and OpenMP implementations of Conway’s game of life.

**MPI** two-sided and version shows a 10% larger loss of efficiency than **MPI** shared implementation when going from 8 to 32 **PE**.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Speedup</th>
<th>Speedup efficiency</th>
<th>Mean (μ)</th>
<th>Standard deviation (σ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI Two-Sided (PE 8)</td>
<td>7.0</td>
<td>87.5%</td>
<td>5.98s</td>
<td>0.00s</td>
</tr>
<tr>
<td>MPI Two-Sided (PE 32)</td>
<td>19.1</td>
<td>59.7%</td>
<td>2.19s</td>
<td>0.00s</td>
</tr>
<tr>
<td>MPI Shared (PE 8)</td>
<td>5.7</td>
<td>71.6%</td>
<td>7.31s</td>
<td>0.02s</td>
</tr>
<tr>
<td>MPI Shared (PE 32)</td>
<td>17.6</td>
<td>54.7%</td>
<td>2.39s</td>
<td>0.03s</td>
</tr>
<tr>
<td>MPI Shared no-init (PE 8)</td>
<td>5.7</td>
<td>71.4%</td>
<td>7.38s</td>
<td>0.00s</td>
</tr>
<tr>
<td>MPI Shared no-init (PE 32)</td>
<td>10.7</td>
<td>33.5%</td>
<td>3.93s</td>
<td>0.25s</td>
</tr>
<tr>
<td>OpenMP (PE 8)</td>
<td>4.6</td>
<td>57.8%</td>
<td>9.12s</td>
<td>2.18s</td>
</tr>
<tr>
<td>OpenMP (PE 32)</td>
<td>10.5</td>
<td>32.9%</td>
<td>4.00s</td>
<td>0.01s</td>
</tr>
</tbody>
</table>

Table 5.7: The mean time, variance, speedup and efficiency for MPI two-sided, MPI shared and OpenMP implementations of Conway’s game of life. **Strong** was used as the dataset and the tests were run on Tegner.

**Weak Scaling**

The base for the scaling factor is the serial code for **MPI** two-sided using the first data point in **weak**. In Fig. 5.12 there can still be seen that the shared implementation start below 80% but has a better curve when using a higher number of **PE**.
5.4. PERFORMANCE COMPARISON OF MPI SHARED, MPI TWO-SIDED AND OPENMP IMPLEMENTATIONS

Figure 5.12: Weak scaling measurement for MPI two-sided, MPI shared memory and OpenMP implementation of Conway’s game of life on Beskow.

The serial implementation was used as the benchmark for MPI weak scaling for the first data point in weak_B. In Fig. 5.12 there can still be seen that the shared implementation starts below 80% but has a better curve when going for higher numbers of PE. The weak scaling and the strong scaling as can be seen in Table 5.8. The weak scaling is slightly lower than the strong scaling for both MPI shared and MPI two-sided. In Fig. 5.12 the weak efficiency is presented.

<table>
<thead>
<tr>
<th></th>
<th>MPI Shared Weak</th>
<th>MPI Shared Strong</th>
<th>MPI Two-Sided Weak</th>
<th>MPI Two-Sided Strong</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 PE</td>
<td>78.5%</td>
<td>82.7%</td>
<td>99.6%</td>
<td>92.7%</td>
</tr>
<tr>
<td>2 PE</td>
<td>78.0%</td>
<td>81.4%</td>
<td>96.2%</td>
<td>99.8%</td>
</tr>
<tr>
<td>4 PE</td>
<td>73.4%</td>
<td>78.4%</td>
<td>91.6%</td>
<td>96.1%</td>
</tr>
<tr>
<td>8 PE</td>
<td>65.0%</td>
<td>71.6%</td>
<td>81.3%</td>
<td>87.5%</td>
</tr>
<tr>
<td>16 PE</td>
<td>57.4%</td>
<td>63.1%</td>
<td>62.0%</td>
<td>68.2%</td>
</tr>
<tr>
<td>32 PE</td>
<td>49.1%</td>
<td>54.7%</td>
<td>55.3%</td>
<td>59.7%</td>
</tr>
</tbody>
</table>

Table 5.8: The weak and strong scaling efficiency for Conway’s game of life that was produced by MPI shared and MPI two-sided.
CHAPTER 5. RESULTS

Profiling Data

The CrayPat profiler was used with the same implementations and configuration as in subsection 4.5.1. The specific CrayPat runtime variable that had been configured to anything other the default was to set PAT_RT_SUMMARY to 0 and PAT_RT_PERFCTR to 10. In Fig. 5.13 and Table 5.9 it can be seen that MPI shared has a larger overhead than MPI two-sided. The High Water Memory Mark is lower for MPI shared than MPI two-sided, showing the shared memory model uses less memory than the message passing model for Conway’s game of life.

![Pie charts showing run time overhead from MPI two-sided and MPI shared](image)

Figure 5.13: Pie chart taken from appearance2 CrayPat to representing the run time overhead from MPI two-sided and MPI shared when using 32 PE on Conway’s game of life.

<table>
<thead>
<tr>
<th></th>
<th>Overhead</th>
<th>Memory Water Mark/PE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI Shared</td>
<td>29.9%</td>
<td>26 MB</td>
</tr>
<tr>
<td>MPI Two-Sided</td>
<td>25.5%</td>
<td>34 MB</td>
</tr>
</tbody>
</table>

Table 5.9: The overhead and High Memory Water Mark from CrayPat for MPI shared and MPI two-sided when using 32 PE on Conway’s game of life.
5.4. PERFORMANCE COMPARISON OF MPI SHARED, MPI TWO-SIDED AND OPENMP IMPLEMENTATIONS

Tegner

The strong scaling tests were performed with the dataset \textit{strong} \_T. The hardware used was a single Haswell node on Tegner, using the Intel compiler and the Intel MPI implementation. Fig. 5.14 presents that the shared implementation has almost the same performance as the two-sided. The standard deviation provides by Table 5.10 give the both of the equivalent good performance. As can be seen, the performance has a better efficiency at max cores than Beskow.

![Graph](image)

Figure 5.14: Speedup (left) and strong scaling (right) for the Conway’s game of life using MPI two-sided and MPI shared memory on Tegner.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Speedup</th>
<th>Speedup Efficiency</th>
<th>Mean ($\mu$)</th>
<th>Standard deviation ($\sigma$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI Two-Sided (PE 8)</td>
<td>6.7</td>
<td>83.0%</td>
<td>6.05s</td>
<td>0.12s</td>
</tr>
<tr>
<td>MPI Two-Sided (PE 24)</td>
<td>15.3</td>
<td>63.9%</td>
<td>2.62s</td>
<td>0.22s</td>
</tr>
<tr>
<td>MPI Shared (PE 8)</td>
<td>6.8</td>
<td>84.2%</td>
<td>5.95s</td>
<td>0.10s</td>
</tr>
<tr>
<td>MPI Shared (PE 24)</td>
<td>15.8</td>
<td>66.0%</td>
<td>2.50s</td>
<td>0.04s</td>
</tr>
</tbody>
</table>

Table 5.10: The mean time, variance, speedup and efficiency for MPI two-sided and MPI Shared implementations of Conway’s game of life. \textit{Strong\_t} was used as dataset and the tests were run on Tegner.
Chapter 6

Analysis and Conclusion

In this chapter, we present the conclusion drawn by the result, the limitations of the evaluation and also the possible further work.

6.1 Analysis

In this section, the results from the matrix-matrix multiplication and Conway’s game of life will be analyzed. Furthermore, a presentation of advantages and disadvantages of using MPI shared in regards to OpenMP and MPI two-sided will be presented.

6.1.1 Allocation of memory

When investigating both of the applications, it can be seen that there is a decrement in the efficiency when the NUMA architecture is not taken into consideration. As can be seen, in Fig. 5.5 and 5.4 there is a substantial drop in performance when needing to access data from the other NUMA node. Even though, a PE is the owner of a virtual memory area the implementations does not populate the memory. In the case of the contiguously allocated memory, this is a good thing. But in the case of the non-contiguous memory allocation there are fewer arguments to use the first touch policy. It is not intuitively but MPICH and OpenMPI implement the shared memory windows with the first touch policy. The point is that there is more important to initializing the memory in an optimal way. In a way using the first touch policy might be for the best. The first touch policy lets the programmer have more freedom instead of pushing them into a corner.

6.1.2 Matrix-Matrix Multiplication

For the matrix-matrix multiplication implementations with a shared memory model the performance curves look similar, as can be seen in Fig. 5.9. The OpenMP implementation is slightly faster than the implementation of MPI shared. For the MPI two-sided implementation, there is no clear evidence why the implementation
drops in performance for 16 PEs and then receives a boost again when using 32 PEs. The gap between the MPI two-sided implementation and the shared memory implementations look reasonable when going from using a single NUMA node to using both NUMA nodes. In Fig. 5.6 the difference is about 10% when utilizing half of the memory on each node, which is the same difference between the message passing implementation (MPI two-sided) and the shared implementations (MPI shared and OpenMP) in Fig. 5.9. CrayPat uses all the memory virtually mapped to a process. Therefore, it could be a difference in how malloc handles the allocation of memory and MPI shared. Also, we know from [33] that MPI two-sided used huge pages and MPI shared is not by default. To this, the usage of memory is based on processes level and because the OpenMP threaded model does not use more than a process, the threaded model could be less memory consuming. For 32 PEs the OpenMP is better in both memory consumption and performance than MPI shared.

6.1.3 Conway’s Game of Life

Because of the variance of the measurements for the OpenMP version, there is a problem evaluating the performance. If the problem is based around sleeping and awaking threads could not be distinguished from the evaluation performed with the profiler. Even if the implementation has around the same performance as MPI shared non-initialized memory, it should still be possible to use the static schedule to create the same parallel split of data between rounds. As the implementation is performed with different compilers the gap between the different MPI implementations disappears when using the Intel compiler. This is important, due to applications outside of HPC does not use the cray compiler. The implementation could have different performances on a different system, in Fig. 5.14 it can be seen that MPI shared out-performance MPI two-sided on the Tegner system.

6.1.4 MPI Shared vs OpenMP

Both shared memory models were faster for one application each. When OpenMP was the winner MPI shared was not very far-off in both performance and memory consumption at 32 PEs. But what can be said about this application is that it is very easy to create parallelism with OpenMP. It is only to adding two rows of #pragmas directives and then the serial program has become a parallel one. There are many tricks to create better performance than the one we had in this thesis but the goal was not to create the fastest implementation. Also as can be seen in section 4.1.3 there is only one section of the code that should be parallelized. OpenMP can do one fork, work and then join before the program is finished. The impact of the NUMA node is found due to wanting to distribute the data between the both nodes. In Conway’s game of life, we saw the improvement when using the NUMA nodes. There is surely also a way of utilizing them with OpenMP but it was hard to control the threads when using #pragmas and the memory is hidden behind an abstraction. There are some advantages with having this abstraction but
6.2 Ethics, Social Aspects and Sustainability

when memory is of most importance it could be good to have more control as MPI shared provides.

6.1.5 MPI Shared vs MPI Two-Sided

From this evaluation, it is clear that the MPI two-sided has a better performance than MPI shared. From the matrix-matrix multiplication evaluation, it can be seen that it is better to have the data in the PE's own local address space. Furthermore, having half of the data on the non-preferred NUMA node has a larger effect than communicating the data between the processes. This is probably due to the possibility to overlap communication with computation. The optimization of the shared memory model is lower in the Cray compiler as seen in section 5.3. This could be to that the library is not optimally implemented and that the library tries to optimize other function like the send and recv functionality of the message passing model. On the other hand, the memory consumption of MPI two-sided is 83% higher than the one of MPI shared, for 32 PEs. The memory consumption of an application is of most importance in current parallel libraries [28]. If there is a check for performance and for memory consumption we would use the shared memory model on each NUMA node and use message passing between the NUMA nodes. After these evaluations are finished we would say that it is even more important to correctly use the hardware setup.

6.2 Ethics, Social Aspects and Sustainability

In this thesis, there are not any ethical viewpoints that can be taken into consideration. Hence, there is just an evaluation of a new way of programming shared memory systems. Furthermore, there are also no real social aspects that can be taken in this thesis. This thesis is mainly contributing to the researching community of parallel programming. On the other hand, this thesis is clearly working towards reducing the resources used for HPC applications. By reducing the memory consumption, there is a possibility to provide a more sustainable parallel application.

6.3 Conclusion

In summary, the main conclusions of this thesis are:

- The two benchmark applications written using MPI shared programming have an execution time that is smaller than OpenMP implementations and comparable to the two-sided implementation.
- MPI shared programming has a memory consumption that is considerably less than MPI two-sided programming and comparable to the OpenMP memory consumption.
• MPI shared programming is a promising and valuable approach to program
shared memory systems.

We compared the performance results with MPI two-sided and OpenMP im-
plementations: applications using MPI shared have speedup that are comparable
to speedup obtained by the applications MPI two-sided communication and signif-
icantly better than the performance of the OpenMP application. The performance
difference between OpenMP and MPI shared is 70% for Conway’s game of life. The
use of MPI shared leads to a considerable decrease of memory consumption in MPI:
a decrease of 45% was observed in the case of the matrix-matrix multiplication ap-
plication. For the GCC and Intel compiler, the shared memory outperforms the
message passing model. With the results presented in this thesis, we extend
the already strong evidence of a possibility of using MPI for both intra communica-
tion and intercommunication on high performance computers. Mapping virtual
memory to a specific NUMA domain can lead to an increment in execution time of
64%.

With the easy usage of both exposing memory to all processes and using the
memory in a fashionable way, MPI clearly provides an applicable utilization of a
shared memory model.

6.4 Limitations of this Work

In the thesis, there are three main limitations:

• Computer architecture
• Applications used in the benchmark
• Dataset of the experiments

The experiments are only carried out on a single type of hardware, the Intel Haswell
node. We could have used other computer architectures in additional to only this
one. During the choice of applications, we came to the conclusion that the thesis did
not have enough time to also include a communication heavy application. A graph
type algorithm could have been a good complement to the other two applications as
it is communication heavy. For the experiment, only one size of data was used and
this leads to not being able to evaluate the impact of other data configurations.

6.5 Future Work

In this thesis, we evaluated the MPI shared for intranode communication. To ac-
tually evaluate if it can be usable as an alternative to today’s implementation of
hybrid models [MPI–OpenMP] it is needed to evaluate if the shared memory model
is usable together with the message passing model. Probably there is some over-
head from the hybrid model as stated in [33], but how big these overheads are
6.5. FUTURE WORK

are left as further work. Also, if the memory consumption is low enough to go over from a pure MPI two-sided implementation to an MPI+MPI implementation is left as further work. Another thing that would need further investigation is if MPI_Comm_split_type would be able to split on a NUMA node basis instead of on only shared memory basis, due to the loss of efficiency when not using the memory optimally.
References


REFERENCES


REFERENCES


Appendix A

Code

The implementations used in this master thesis are available at: https://www.dropbox.com/sh/tb1urxe1u1915h6/AACTv4eWAtAdvfpAmDKQna?dl=0
If for some reason the code is not available at the URL, email: david.karlbom@gmail.com