***Comparative Analysis of NLP and OpenAI-based Chatbots: A User-Centric Evaluation***

**Vignesh Suvarna**

School of Computing Science

Sir Alwyn Williams Building

University of Glasgow

G12 8QQ

A dissertation presented in part fulfilment of the requirements of the

Degree of Master of Science at The University of Glasgow

August 2023

**Abstract**

The creation and implementation of chatbots have emerged as essential tools for seamless communication and information exchange in a time when human-computer interaction is quickly evolving. In this paper we discuss the development and compare the performance of two different chatbot models: an OpenAI-powered chatbot that uses sophisticated language models and an NLP-based chatbot that uses traditional NLP methods. The objective is to assess their respective performances and user preferences through a comprehensive user survey.

The study involves the design and development of both chatbots, each representing a different approach to conversational AI. The subsequent evaluation encompasses response accuracy, contextual understanding, and user satisfaction. A user survey gathers insights on clarity, coherence, usefulness, and overall experience.

This research examines the advantages and disadvantages of each strategy by contrasting the capabilities of state-of-the-art AI-driven chatbots with more conventional NLP-driven analogues. The research provides developers and organisations with subtle insights that may be used to make decisions that are appropriate for a certain use case. This study provides relevant insights into the changing chatbot technology environment as AI and NLP transform human-computer interaction. As AI and NLP continue to reshape human-computer interaction, this dissertation serves as a timely and insightful contribution to the ongoing discourse in the field.

**Education Use Consent**

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. Please note that you are under no obligation to sign this declaration, but doing so would help future students.

Name: Vignesh Suvarna Signature: Vignesh Suvarna

**Acknowledgements**

At the special moment of completing this dissertation paper, the one-year master’s life at the University of Glasgow is coming to an end. I experienced a wonderful time living and studying at the University of Glasgow.

I like to take this opportunity to express my sincerest gratitude to my supervisor, classmates and friends who have helped me.

Thanks to my supervisor, a patient and responsible professor. He has given a lot of guidance in the paper’s topic selection, conception and writing. The completion of this project is inseparable from my supervisor’s direction and discussion.

Thanks to my classmates and friends for helping me learn to use standard tools to write papers and reminding me of many precautions when I am busy. The on-time completion of this paper is inseparable from your patient reminders.

Finally, I would like to thank my family and girlfriend for their great help and care during my postgraduate life, making my life full of hope and vigour.

”The road ahead will be long. Our climb will be steep.” I will use this poem to spur myself to overcome obstacles and forge ahead on the road of life in the future.

Contents

[1 Introduction 5](#_Toc49458)

[2 Background 6](#_Toc49459)

[2.1 Numerical Weather Prediction 6](#_Toc49460)

[2.2 Successive Over-Relaxation Algorithm 6](#_Toc49461)

[2.3 Message Passing Interface 8](#_Toc49462)

[2.4 ArgoDSM 9](#_Toc49463)

[3 Design & Implementation 10](#_Toc49464)

[3.1 Data dependency analysis 10](#_Toc49465)

[3.2 Design of SOR Algorithm Based on MPI 11](#_Toc49466)

[3.3 Implementation of SOR Algorithm Based on MPI 12](#_Toc49467)

[4 Testing & Evaluation 14](#_Toc49468)

[4.1 Testing environment 14](#_Toc49469)

[4.2 Testing procedures 15](#_Toc49470)

[4.2.1 Summarize testing parameters 15](#_Toc49471)

[4.2.2 Deploy testing instances 16](#_Toc49472)

[4.3 Testing result evaluation 16](#_Toc49473)

[4.3.1 The effect of compiler optimization flag 16](#_Toc49474)

[4.3.2 The overall performance of each implementation 18](#_Toc49475)

[4.3.3 The SOR performance on fixed array volume 19](#_Toc49476)

[4.3.4 The effect of node count on MPI and ArgoDSM 20](#_Toc49477)

[4.3.5 The effect of dsmNX and dsmNY on ArgoDSM 22](#_Toc49478)

[5 Conclusion & Future Work 23](#_Toc49479)

**Chapter 1**

**Introduction**

In an era defined by rapid technological evolution, the convergence of artificial intelligence (AI) and natural language processing (NLP) has led to the proliferation of chatbots, revolutionizing the way humans interact with machines. This paper launches thorough research into the capabilities of two distinctive paradigms within the chatbot landscape: the cutting-edge OpenAI model and a Natural Language Processing (NLP) model powered by TensorFlow, a flexible machine learning framework. Chatbots have developed in the modern environment from simple scripted responses to dynamic beings capable of contextual comprehension and natural interaction. These virtual companions now have a wide range of uses, including customer service in e-commerce, smartphone virtual assistants, and even mental health care. This growth underscores their crucial function in speeding information transmission and improving user experiences in a variety of industries.

Chatbots have orchestrated a transformative shift in the internet landscape, ushering in a new era of interactive digital experiences. By transcending traditional static web interfaces, these intelligent agents have made it possible for dynamic, real-time discussions to mimic human interactions. They have a significant positive effect on people, improving customer service, boosting information accessibility, and personalising user experiences. Chatbots have increased user engagement by promoting quick responses and cutting down on wait times, providing a seamless link between users and digital services. As virtual companions, they have improved operational effectiveness while also changing how people interact with technology, making the internet more user-friendly, approachable, and responsive.

In the fields of artificial intelligence (AI) and machine learning, OpenAI is a well-known research institution. They have significantly contributed to creating sophisticated language models like GPT. These models are designed to understand and generate human-like text by training on vast amounts of diverse textual data.

Natural Language Processing (NLP) makes it possible for computers to meaningfully and accurately read, interpret, and produce human language. NLP techniques span a wide range of methodologies, including rule-based systems and machine learning-based approaches. Language translation, sentiment analysis, text parsing and analysis to extract information, among many other things, are some of these techniques. TensorFlow, a cornerstone of modern machine learning, stands as a vital enabler, facilitating the creation of sophisticated neural architectures that underpin the NLP model's ability to discern language nuances and formulate coherent responses.

This research aspires to decipher the capabilities and constraints of two divergent paths of chatbot development: the forefront of OpenAI's innovation and the functional prowess of NLP techniques bolstered by TensorFlow. By meticulously evaluating their performances and soliciting user perspectives, this study not only illuminates their inherent potential but also navigates the complex interplay between innovation and established methodologies.

The subsequent chapters of the dissertation are as follows:

* Chapter 2
* Chapter 3

**Chapter 2**

**Background**

2.1 Numerical Weather Prediction

The process of Twined Buffering solution is described as follows.

Algorithm 2 SOR Boundary Processing

Input: p0, p1, i, j, k

Output: p1 index, p1 value if i == im + 1 then

p1 index ← F3D2C(i, j, k) p1 value ← p0[F3D2C(i - im, j, k)]

else if i == 0 then

p1 index ← F3D2C(i, j, k) p1 value ← p0[F3D2C(i + im, j, k)]

else if j == jm + 1 then

p1 index ← F3D2C(i, j, k) p1 value ← p0[F3D2C(i - 1, j, k)]

else if j == 0 then

p1 index ← F3D2C(i, j, k) p1 value ← p0[F3D2C(i, j, k)]

end if

Algorithm 3 SOR Core Logic

Input: p0, p1, i, j, k

Output: p1 index, p1 value if k == 0 or k == km + 1 then

raise error(’not a core position’)

end if cn1 ← 1.0 / 3.0 cn2l, cn2s, cn3l, cn3s, cn4l, cn4s ← 0.5 omega ← 1.0 itmp ← cn2l × p0[F3D2C(i + 1, j, k)] + cn2s × p0[F3D2C(i - 1, j, k)] jtmp ← cn3l × p0[F3D2C(i, j + 1, k)] + cn3s × p0[F3D2C(i, j - 1, k)] ktmp ← cn4l × p0[F3D2C(i, j, k + 1)] + cn4s × p0[F3D2C(i, j, k - 1)] sumtmp ← cn1 × (itmp + jtmp + ktmp - rhs[F3D2C(i, j, k)]) - p0[F3D2C(i, j, k)] p1 value ← p0[F3D2C(i, j, k)] + omega × sumtmp p1 index ← F3D2C(i, j, k)

1. Define the shape of the three-dimensional array to be computed as (im, jm, km).
2. Use the initial data of p0, p1, rhs to create three three-dimensional arrays with same shape. The shape of the array is (im+2, jm+2, km+2).
3. From the first iteration: in the odd iteration, p0 is used as the reference array and p1 as the updated array; in the even iteration, p1 is used as the reference array and p0 as the updated array.
4. Repeat until all iterations are completed.

The internal logic within each iteration is described as pseudo code. The logic of boundary processing is shown in algorithm. The core logic of SOR is shown in algorithm 3.

## 2.3 Message Passing Interface

MPI (Message Passing Interface) is a cross-language communication protocol for writing parallel computing programs.[9] As an information transfer application program interface, MPI includes protocols and semantic descriptions, which indicate how to play relevant characteristics in various implementations. Supporting for point-to-point communication and broadcast communication, MPI can achieve high performance, large-scale, and portability.[7] MPI is currently the primary model of high-performance computing.

In MPI, each node is called a communicator, abbreviated as COMM. For any MPI calculation task: the total number of nodes is called world communicator size, abbreviated as WORLD COMM SIZE; each node has a unique index number (increasing from 0), called world communicator rank, abbreviated as WORLD COMM RANK.

When using MPI to write parallel operations, the nodes of different WORLD COMM RANK are generally assigned specific calculation tasks according to WORLD COMM SIZE. When communication between computing nodes is needed, we can use MPI SEND to send data to the specified node, or use MPI RECV to receive data from the specified node. In order to avoid deadlock caused by the logical confusion of sending and receiving, MPI also provides an interface like MPI Sendrecv to send and receive data in pairs.

Algorithm 4 Generic MPI Model

Input: An array of input data

Output: An array of output data mpi init()

world rank ← mpi comm rank() world size ← mpi comm size()

output

data

←

customized

strategy(world

rank, world

size, input

data)

mpi

finalize()

When the computing task involves the master-slave model, MPI also provides MPI BCAST (master node broadcast data), MPI SEND, MPI RECV (master-slave node intercommunication data) such operation paradigms to simplify the logic of the code.

Algorithm 5 Master-Slave MPI Model

Input: An array of input data

Output: An array of output data mpi init()

world rank ← mpi comm rank() world size ← mpi comm size() if world rank ̸= 0 then

worker strategy(world rank, world size)

else output data ← manager strategy(world size, input data)

end if mpi finalize()

## 2.4 ArgoDSM

ArgoDSM is a modern distributed shared-memory system for high-performance computing and big data. Consistent global address space can support large-scale shared-memory programming in a distributed system.[2] ArgoDSM (the shared-memory system) can support large-scale operations without this scale’s dedicated hardware.

Traditional consistency methods are often achieved through centralized active master node catalogues). If the computing node can involve as little as possible communication with the remote node (or even other sockets) and make decisions locally, it can avoid the long delay caused by the network and the software message processing program. ArgoDSM has a mechanism to pre-fetch data across computing nodes to enable parallel computing and data exchange within a single node.

Chapter 3

# Design & Implementation

This chapter will first use the serial SOR algorithm mentioned in section 2.2 as a benchmark to design and implement a MPI-based SOR parallel algorithm.

## 3.1 Data dependency analysis

Before designing the parallel algorithm, first analyze the data dependence relationship in the serial algorithm, and find out the data items that do not depend on each other as the design basis of the parallel algorithm. It can be seen from Section 2.2 that the serialized SOR algorithm will traverse the three dimensions i, j, and k sequentially, where i is the outermost loop. Analyze the data dependencies in the sequential traversal process as shown in figure 3.1.

The red dashed arrow indicates the dependence of the data on the corresponding position of the p0 array, applied to any plane formed by the i-axis and j-axis.

The solid yellow arrow indicates the dependence of the data on the position of the p0 array corresponding to (i, j, k - 1), applied to any plane formed by the i-axis and j-axis except for the ones that k == 0 or k == km + 1.

The solid orange arrow indicates the dependence of the data on the position of the p0 array corresponding to (i, j, k + 1), applied to any plane formed by the i-axis and j-axis except for the ones that k == 0 or k == km + 1.

The light blue dot indicates the dependence of the data on the exact position of the p0 array.

The dark blue dot indicates the dependence of the data on the exact position of the rhs array.

Obviously, in the k-axis direction, the SOR algorithm only depends on the data of its adjacent layer.

Therefore, splitting and distributing the parallel data array in the k-axis direction can achieve a better locality.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  | *jm+1 jm jm-1*  *…*  *2*  *1*  *0* |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

*jm+1*The value depends

on another one in p0

array at the end of the

*jm*arrow.

*jm-1*The value depends

on another one in p0

array at the end of the

*…*arrow (i, j, k - 1).

*2*

The value depends

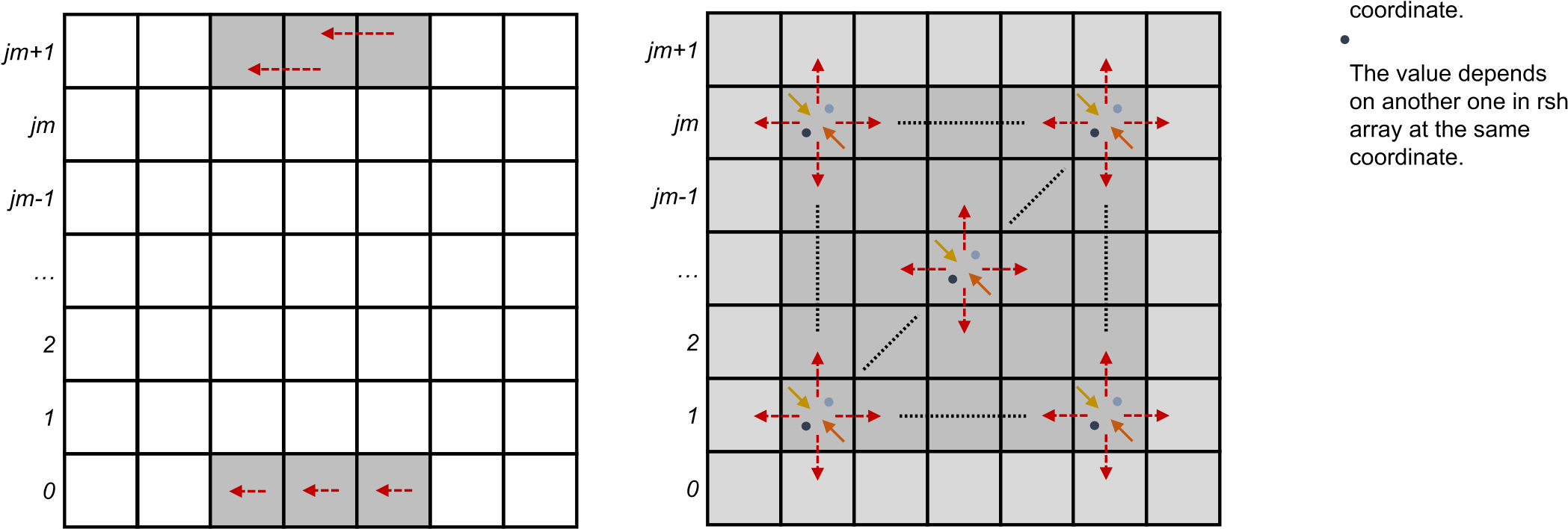
*1*on another one in p0

array at the end of the arrow (i, j, k + 1).

*0*

*0 1 2 … im-1 im im+1 0 1 2 … im-1 im im+1*

The value depends on another one in p0 array at the same



*0 1 2 … im-1 im im+1 0 1 2 … im-1 im im+1*

Figure 3.1: The data dependencies of SOR algorithm

## 3.2 Design of SOR Algorithm Based on MPI

The analysis in section 3.1 shows that the Twined Buffering SOR solution has excellent locality inside the plane formed by the i-axis and j-axis. Therefore, when designing the MPI-based SOR parallel algorithm, the data should be divided into several cubes alongside the i-j plane, based on the height of the k-axis, and distributed to each parallel computing node.

The parallel SOR algorithm based on MPI is described as follows.

Assuming that the SOR input array shape is (im, jm, km), the number of parallel computing nodes is node count. Define the three input array as p0, p1 and rhs.

For each computing node:

1. Obtain its world size and world rank by calling the API in MPI libraries.
2. Calculate the range of input array related to itself by algorithm 6. The range is described as a

cube restricted by ((*imin,imax*)*,*(*jmin,jmax*)*,*(*kmin,kmax*)).

Algorithm 6

Get Input Array Range

Input:

im, jm, km, world

size, self

world

rank

Output:

i

min, i

max, j

min, j

max, k

min, k

max

i min ← 0 i max ← im + 1 j min ← 0 j max ← jm + 1

base k size = (km + 2) / world size base k rem = (km + 2) % world size if world rank == 0 then k min ← 0

k max ← base k size + base k rem

else

k min ← self world rank × base k size + base k rem k max ← k min + base k size - 1

end if

1. Initialize p0, p1 and rhs by the range of input array.
2. In each iteration, Send the data of the uppermost i-j plane and the lowermost i-j plane to the neighbor nodes, and receive the data of the adjacent i-j plane from the neighbor nodes. The calculation method of neighbor nodes rank is described in algorithm 7.
3. In each iteration, complete the actual SOR computation according to the algorithm.

else

Algorithm 7

Get Neighbor Ranks

Input:

world

size, self

world

rank

Output:

lower

neighbor, upper

neighbor

if

world

rank

*>*

0

then

lower

neighbor

←

self

world

rank - 1

lower

neighbor

←

mpi

proc

null

end if

if world rank *<* world size - 1 then

upper neighbor ← self world rank + 1 else

upper neighbor ← mpi proc null

end if

## 3.3 Implementation of SOR Algorithm Based on MPI

The implementation of SOR Algorithm based on MPI is written in C++ Language. The flow chart of the program is shown in figure 3.2.

Start

MPI\_Init

MPI\_Finalize

Get Context

input\_array\_range

mpi\_world\_size

mpi\_world\_rank

Initialize array p0, p1, rhs

Iteration Finished?

Collect Data

Y

Find Neighbor Process

N

Transfer boundary data to neighbor process

Parallel

Receive boundary data from neighbor process

Wait

Solve SOR Boundary Conditions

Solve SOR Core Logic

End

Figure 3.2: The flow chart of SOR Implementation based on MPI

Chapter 4

# Testing & Evaluation

## 4.1 Testing environment

To make the results reproducible, all of the tests are initiated on a cloud virtual machine. Table 4.1 and 4.2 shows the hardware and software environments, respectively.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | The Testing Hardware Environment |  |  |
|  |  | CPU |  | Memory |
| Model Name | Cores | Siblings Clock Speed Cache Size | Size | Bandwith |
| Intel(R) Xeon(R) | 8 | 16 3.10GHz 25344 KB | 64 GiB | 5618.622 MiB/s |

Table 4.1: The Testing Hardware Environment

The CPU is an Intel(R) Xeon(R) model that clocks at 3.10 GHz with hyper-threading enabled. The memory has an maximum capacity of 64 GiB with 5618.622 MiB/s MEMCPY Bandwith[[1]](#footnote-1).

|  |  |
| --- | --- |
|  | The Testing Software Environment |
| OS | Linux version 5.4.0-1057-gcp (buildd@lgw01-amd64-035) (Ubuntu 18.04) |
| make | GNU Make 4.1, Built for x86 64-pc-linux-gnu |
| cmake | cmake version 3.22.0 |
| ldd | ldd (Ubuntu GLIBC 2.27-3ubuntu1.4) 2.27 |
| gcc | gcc (Ubuntu 7.5.0-3ubuntu1∼18.04) 7.5.0 |
| g++ | g++ (Ubuntu 7.5.0-3ubuntu1∼18.04) 7.5.0 |
| mpicc | gcc (Ubuntu 7.5.0-3ubuntu1∼18.04) 7.5.0 |
| mpic++ | g++ (Ubuntu 7.5.0-3ubuntu1∼18.04) 7.5.0 |
| mpicxx | g++ (Ubuntu 7.5.0-3ubuntu1∼18.04) 7.5.0 |
| mpirun | mpirun (Open MPI) 2.1.1 |
| ArgoDSM | CommitID = 4a7789af174aeeba1c75cb589a1676dd238872a1 |

Table 4.2: The Testing Software Environment

The operating system is Ubuntu 18.04 built upon Linux kernel version 5.4.0-1057-gcp. Swap memory is disabled 2 to improve the accuracy of results. MPI utilities are installed from pre-built release[[2]](#footnote-2). ArgoDSM is built and installed from source4.

## 4.2 Testing procedures

### 4.2.1 Summarize testing parameters

When measuring the performance, to get more scientific and accurate results, we should first summarize the parameters that may affect the computing performance to design a set of test plans based on all the parameters.

The performance of the SOR algorithm is intrinsically related to the size of the three-dimensional data array, that is, three parameters: im, jm, and km (or array shape = (im, im, km) as a combination).

The C reference implementation is a serial computing process. By iterating multiple array shape combinations, serial computing performance under different array shapes can be obtained as reference data for parallel performance measurement.

The most significant parameter in parallel algorithms is the number of parallel nodes participating in the computing (abbreviated as node count).

In MPI, as stated above in Section 3, node count will affect the number of parallel computing tasks. However, due to the complexity of factors such as CPU context switching cost and memory access strategies, the performance of SOR computing may not be positively correlated with node count. Therefore, assuming that the CPU supports *nproc* cores, it would be better to obtain performance data when node count takes values in {*nproc/*2*,nproc/*8*,nproc/*16}.

In ArgoDSM, some additional parameters affect the computing performance: dsmNX, dsmNY, dsmVersion. There is a constraint relationship described by equation 4.1. The possible values of dsmVersion are: 1, 2, 3. Therefore, it would be better to obtain performance data from all possible dsmNX, dsmNY, dsmVersion combinations.

*node count* = *dsmNX* × *dsmNY* (4.1)

Whether to use the compiler optimization flag’-O3’ may also affect the performance of each implementation.

According to the previous analysis in this section, considering the testing environment given in section 4.1, summarize the testing parameters in table 4.3.

|  |  |
| --- | --- |
|  | Testing Parameters |
| Parameter | Values |
| im | 128, 256, 512 |
| jm | 128, 256, 512 |
| km | 64, 128, 256 |
| method | numbers from 0 to 4 represents C, ArgoDSM V1, V2, V3 and MPI |
| optimized | compiler -O3 optimization flag: 0 means not set, 1 means set |
| node count | 2, 8 ,16 |
|  | if node count == 2: (1, 2), (2, 1) |
| dsmNX, dsmNY | if node count == 8: (1, 8), (2, 4), (4, 2), (8, 1) if node count == 16: (1, 16), (2, 8), (4, 4), (8, 2), (16, 1) |

Table 4.3: Testing Parameters

### 4.2.2 Deploy testing instances

Based on the test parameters proposed in section 4.2.1, we will run 1998 single test instances to collect the running time (5 iterations) of the SOR algorithm for each combination of parameters. Note that in each test, the running time of the SOR algorithm should be the longest one among multiple computing nodes.

Due to the large number of testing instances, it is crucial to initiate all tests automatically by a Python script. Then obtain the time consumed by the SOR algorithm by parsing the original log files generated during the test.

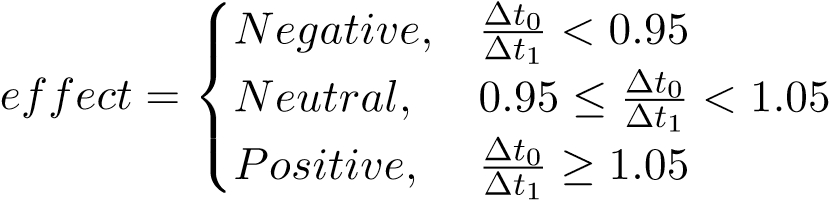
The scripts for deploying testing instances and collecting data are accessible on GitHub[[3]](#footnote-3). The original 1998 testing log files and testing results are also available on GitHub[[4]](#footnote-4).

## 4.3 Testing result evaluation

### 4.3.1 The effect of compiler optimization flag

The compiler used in this test can optimize the code at the cost of longer compilation time. Usually, such optimization can improve the efficiency of the program. In order to verify whether this optimization can bring the expected positive impact on each SOR algorithm implementation, it is necessary to compare the code performance before and after the compiler optimization flag is set when the other parameters are precisely the same.

Assuming that the SOR running time before the optimization is ∆*t*0 and the optimized SOR running time is ∆*t*1, define the effect of compiler optimization in equation 4.2.

 (4.2)

Analyze the 1988 test records. Create table 4.4 and area figure 4.1 showing the number of effects brought by the compiler optimization.

The effect of compiler '-O3' optimization flag

C

ArgoDSM V1

ArgoDSM V2

ArgoDSM V3

MPI

0

50

100

150

200

250

300

Number of different effects

Negative Count

Neutral Count

Positive Count

Figure 4.1: The effect of compiler optimization flag

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | The number of effects brought by compiler optimization | | | |  |  |
| Effect | C Reference | ArgoDSM V1 | ArgoDSM V2 | ArgoDSM V3 | MPI | Sum |
| Negative | 0 | 25 | 7 | 5 | 0 | 37 |
| Neutral | 0 | 102 | 141 | 28 | 0 | 271 |
| Positive | 27 | 170 | 149 | 264 | 81 | 691 |
| Sum | 27 | 297 | 297 | 297 | 81 | 999 |

Table 4.4: The number of effects brought by compiler optimization

For the C and MPI implementation, compiler optimization brings a positive impact on all test instances. For the ArgoDSM implementation, the compiler has a positive effect in most cases (65.432%), and only a negative effect in a few cases (4.153%).

According to this conclusion, we can only consider the records with compiler optimization in the subsequent data analysis, safely reducing the number of test records from 1998 to 999.

### 4.3.2 The overall performance of each implementation

Based on the conclusion proposed in section 4.3.1, generate figure 4.2 from 999 optimized test records to analyze the overall performance of each SOR implementation. This bar graph is composed of 27 sets of data, and each set of data represents the performance of five SOR implementation methods under a certain array shape. The performance achieved by each SOR implementation under each fixed array shape is the shortest running time that this code can achieve under all possible combinations of parameters (node count, dsmNX and dsmNY). The vertical axis of this figure uses a logarithmic coordinate with base 10 to obtain a better display effect.

The SOR execution time (seconds) of each implementation in log10 scale

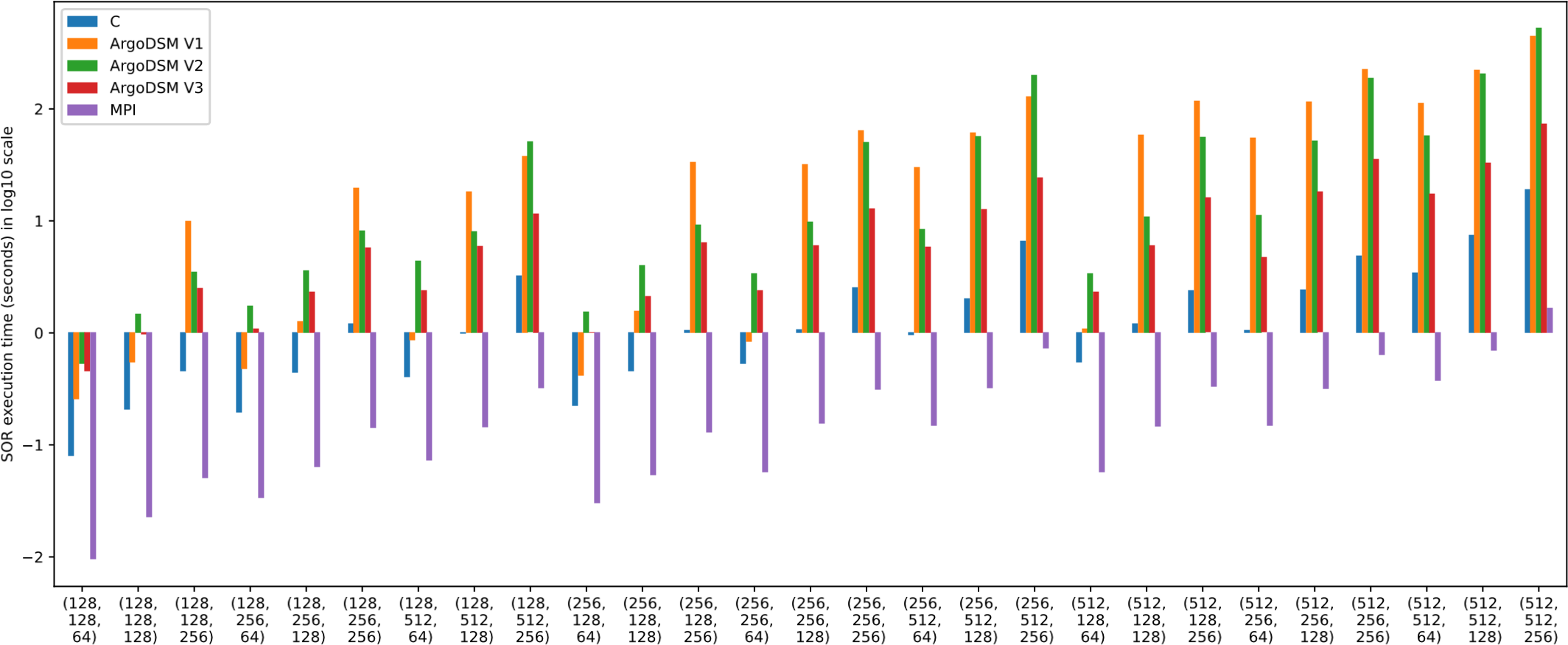


Figure 4.2: The SOR execution time (seconds) of each implementation in log10 scale

By analyzing the figure, the conclusion is as follows.

1. The code based on MPI has a positive acceleration effect compared to the C reference code, under all tested array shapes.
2. Although the CPU in the test environment has 16 cores, the maximum speedup of MPI code compared to C code is only about 10 times.
3. The code based on ArgoDSM has a negative acceleration effect compared to the C reference code, under all tested array shapes.
4. Among the three versions of ArgoDSM, in most cases, version 1 has the worst performance, and version 3 has the best performance.
5. In the context of this paper’s test environment and test vectors, MPI has a better performance than ArgoDSM.

We will analyze the reasons behind conclusion (2) (3) (4) in the following subsections.

### 4.3.3 The SOR performance on fixed array volume

Before analyzing the impact of node count, dsmNX and dsmNY on performance, we need first to figure out the impact of array shape as a context variable.

According to the testing array shapes summarized in section 4.2.1, the values of array volume (*array volume* = *im* × *jm* × *km*) could be: 220, 221, 222, 223, 224, 225 and 226. Generate figure 4.3 from 999 optimized test records to analyze the performance of each SOR implementation under fixed array volumes. In each sub-figure, the SOR execution time can reflect the impact of the variation of the array shape on the SOR execution time when the test vector size is the same.

The SOR execution time (secondes) of each Implementation under fixed array volumes

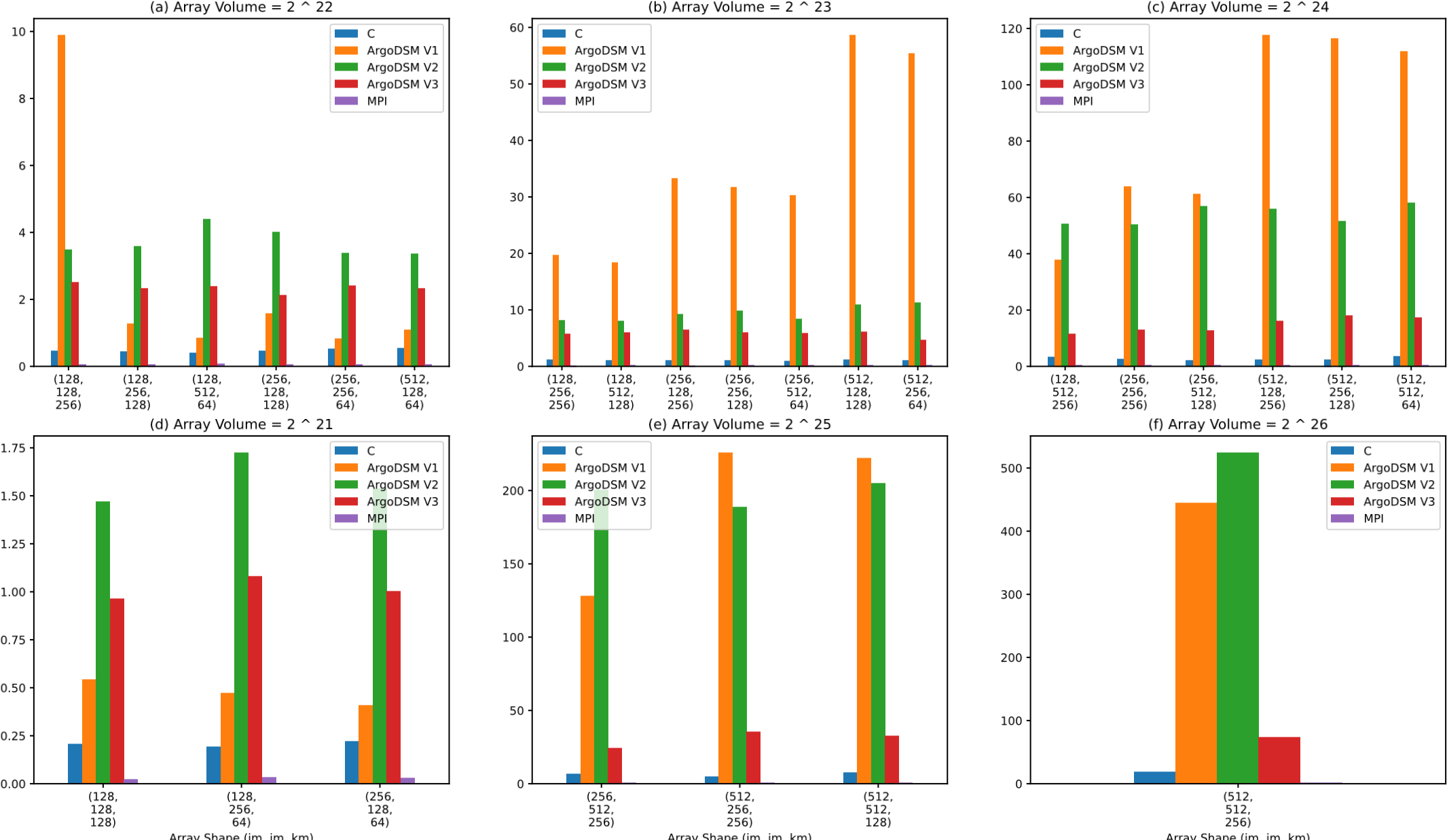


Figure 4.3: The SOR execution time (seconds) of each Implementation under fixed array volumes By analyzing the figure, the conclusion is as follows.

1. The performance of MPI, ArgoDSM V2 and ArgoDSM V3 will only be slightly affected by the variations of array shape if array volume is fixed.
2. According to figure 4.3.(b, c, d), the increasing of im in ArgoDSM V1 will increase the SOR execution time proportionally when array volume is larger equal than 223.

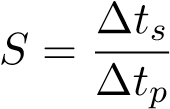
The reason is that, although ArgoDSM Version 1 has a mechanism to linearly pre-fetch data across computing nodes, however, the dependency pattern is non-linear: each of the SOR points relies on the six adjacent points in the direction of i, j and k.

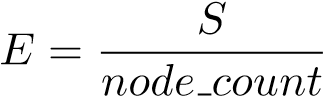
1. According to figure 4.3.(a, d), ArgoDSM V1 has better performance than V2 and V3 when array volume is smaller equal than 222.
2. According to figure 4.3.(a), ArgoDSM V1 has an unexpected performance loss when array shape is (128*,*128*,*256). It always shows the same performance loss in repeated tests.

### 4.3.4 The effect of node count on MPI and ArgoDSM

Based on the conclusion proposed in section 4.3.3, we can analyze the impact of node count on performance in MPI, without worrying about the interference introduced by array shape.

Assuming that the serial SOR running time in C reference is ∆*ts* and the parallel SOR running time in MPI is ∆*tp*. Define speedup ratio (as *S*) and parallel efficiency (as *E*) in equation 4.3 and equation 4.4, respectively.

 (4.3)

 (4.4)

Generate figure 4.4 to analyze the impact of node count on MPI speedup ratio and parallel efficiency.

MPI speedup ratio and parallel efficiency

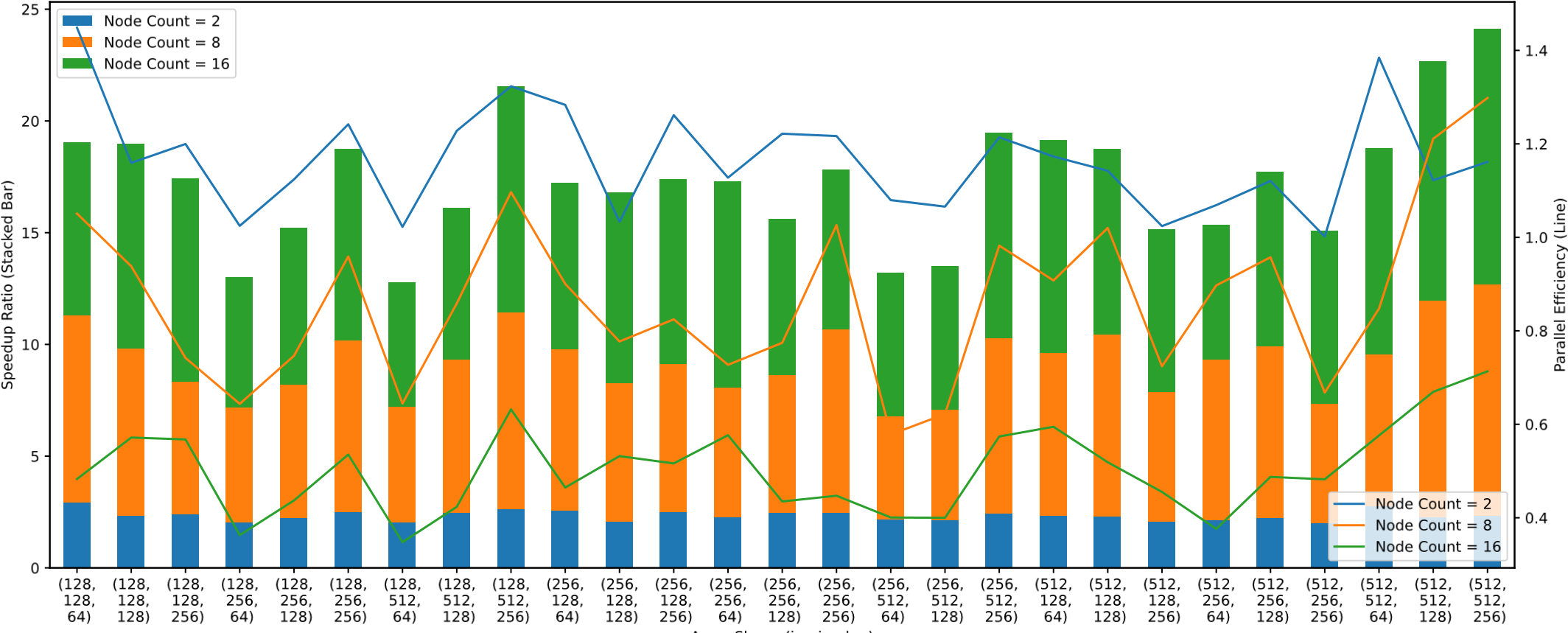


Figure 4.4: MPI speedup ratio and parallel efficiency

The stacked bars represent the speedup ratio of different node count in tested array shapes. The lines represent the parallel efficiency of different node count in tested array shapes.

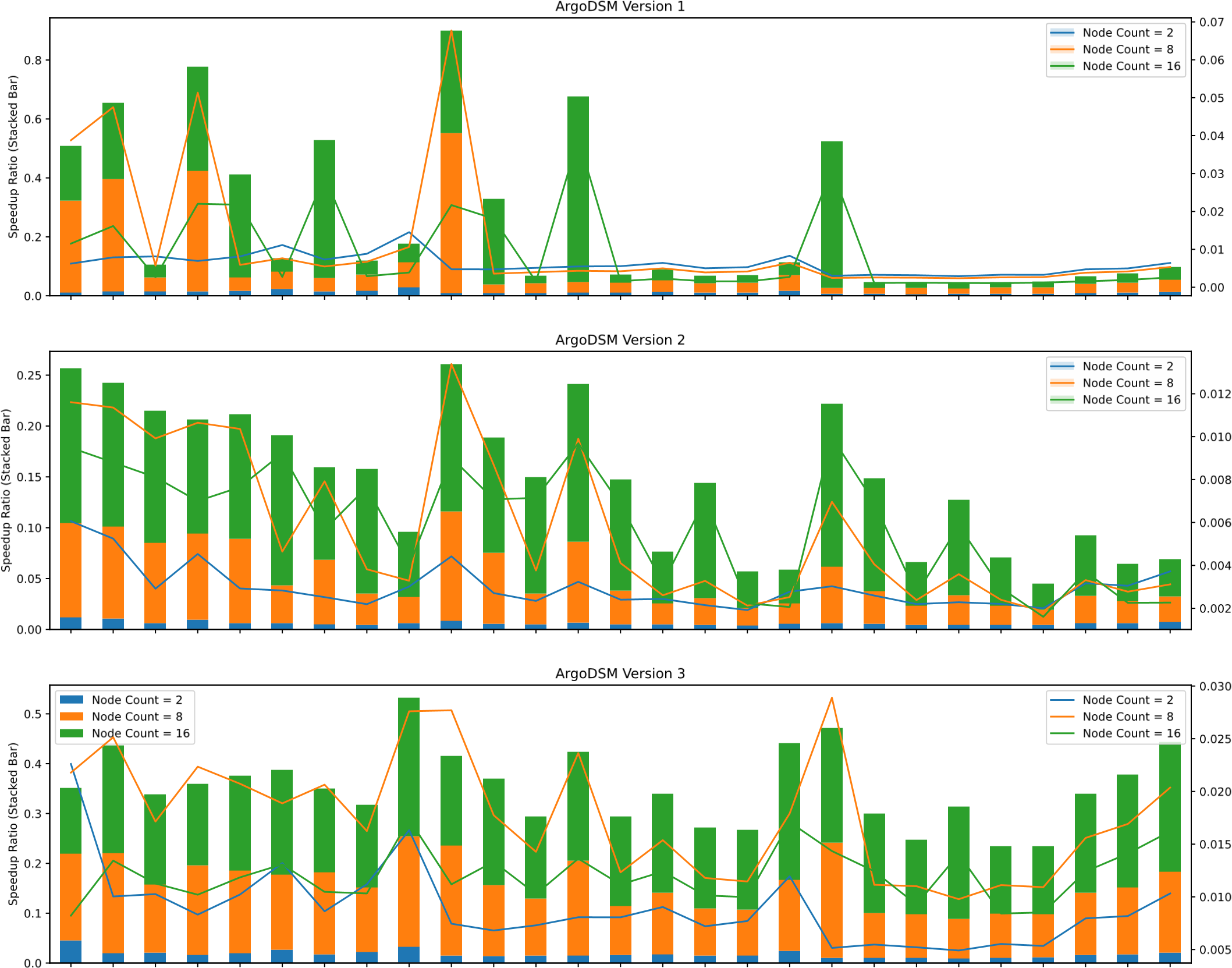
By analyzing figure 4.4, the conclusion for MPI is as follows.

1. The greater the node count, the higher the speedup ratio.
2. The fewer the node count, the higher the parallel efficiency.

The average parallel efficiency is lower than 0.5 when node count = 16. The reason may be related to the CPU cache and memory access mechanism, especially under the fact that the CPU in the test environment has only 8 physical cores, but hyper-threading is enabled to support 16 node count.

1. When im ≤ jm, the parallel efficiency increases with the increase of km.

ArgoDSM speedup ratio and parallel efficiency



(128,(128,(128,(128,(128,(128,(128,(128,(128,(256,(256,(256,(256,(256,(256,(256,(256,(256,(512,(512,(512,(512,(512,(512,(512,(512,(512, 128, 128, 128, 256, 256, 256, 512, 512, 512, 128, 128, 128, 256, 256, 256, 512, 512, 512, 128, 128, 128, 256, 256, 256, 512, 512, 512,

64) 128) 256) 64) 128) 256) 64) 128) 256) 64) 128) 256) 64) 128) 256) 64) 128) 256) 64) 128) 256) 64) 128) 256) 64) 128) 256)

Array Shape (im, jm, km)

Figure 4.5: ArgoDSM speedup ratio and parallel efficiency

Generate figure 4.5 to analyze the impact of node count on ArgoDSM speedup ratio and parallel efficiency. The conclusion for ArgoDSM is as follows.

1. In ArgoDSM V1, the speedup ratio will not change steadily as node count changes.
2. In ArgoDSM V2, the greater the node count, the higher the parallel efficiency.
3. In ArgoDSM V3, the implementation has best overall parallel efficiency when node count =

8.

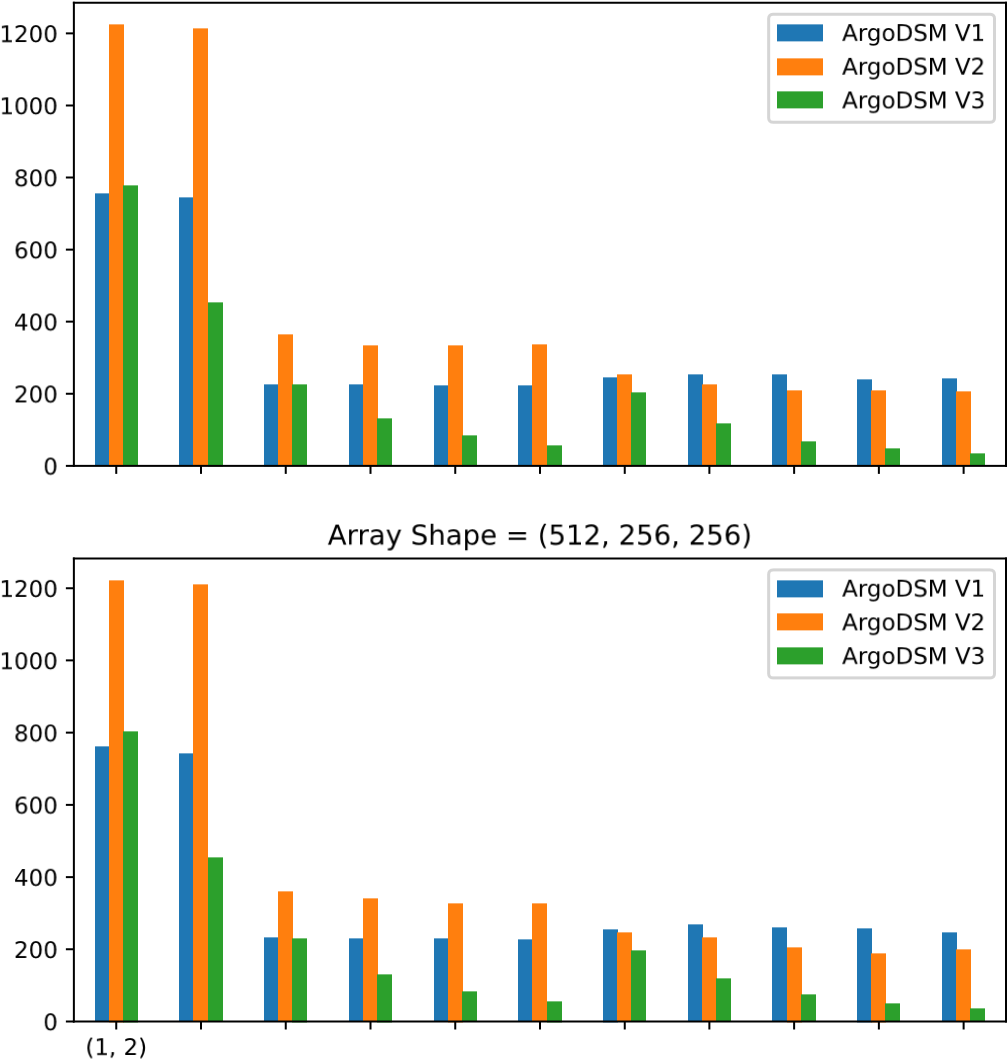
### 4.3.5 The effect of dsmNX and dsmNY on ArgoDSM

Before analyzing the impact of dsmNX and dsmNY on ArgoDSM, we must first pick up the array shapes that can produce stable results.

According to figure 4.5, the following array shapes are suitable for analysis: (512*,*512*,*128), (512*,*512*,*256), (512*,*256*,*256), (256*,*512*,*256).

The SOR execution time (seconds) of ArgoDSM under different (dsmNX, dsmNY)

Array Shape = (512, 512, 128) Array Shape = (512, 512, 256)

(2, 1) (1, 8) (2, 4) (4, 2) (8, 1) (1, 16) (2, 8) (4, 4) (8, 2) (16, 1) (1, 2) (2, 1) (1, 8) (2, 4) (4, 2) (8, 1) (1, 16) (2, 8) (4, 4) (8, 2) (16, 1)

ArgoDSM V1

ArgoDSM V2

ArgoDSM V3

Array Shape = (256, 512, 256)

ArgoDSM V1

ArgoDSM V2

ArgoDSM V3

(dsmNX, dsmNY) (dsmNX, dsmNY)

Figure 4.6: The SOR execution time (seconds) of ArgoDSM under different (dsmNX, dsmNY)

According to figure 4.6, it is obvious that the value of (dsmNX, dsmNY) will not significantly affect the performance under same node count.

Chapter 5

# Conclusion & Future Work

In this paper, we designed and implemented the Successive Over-Relaxation algorithm based on MPI, and evaluated the performance of MPI and ArgoDSM by measuring the SOR computing time. In the context of the testing environment in this paper, MPI has better acceleration effect and more stable performance than ArgoDSM. This paper’s can be used not only to accelerate Numerical Weather Prediction, but also to solve three-dimensional Poisson equations in more general scenarios.

Due to hardware and time constraints, we only evaluated the performance of the SOR algorithm on a single machine, and the size of the test vector is only 256 MiB. The performance evaluation given in this paper is more suitable as a reference for small computing devices such as Raspberry Pi.

In the future, we aim to (1) test each implementation on a large scale computing cluster; (2) increase the size of the test data to 256 GiB to more comprehensively evaluate the computing performance of MPI and ArgoDSM; (3) read the source code of the ArgoDSM framework to research on the mechanism of the unexpected ArgoDSM performance in section 4.3.3.

Bibliography

1. Peter Bauer, Alan Thorpe, and Gilbert Brunet. The quiet revolution of numerical weather prediction. *Nature*, 525(7567):47–55, 2015.
2. Stefanos Kaxiras, David Klaftenegger, Magnus Norgren, Alberto Ros, and Konstantinos Sagonas. Turning centralized coherence and distributed critical-section execution on their head: A new approach for scalable distributed shared memory. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 3–14, 2015.
3. Corey Lesk, Pedram Rowhani, and Navin Ramankutty. Influence of extreme weather disasters on global crop production. *Nature*, 529(7584):84–87, 2016.
4. Edward N Lorenz. Reflections on the conception, birth, and childhood of numerical weather prediction. *Annu. Rev. Earth Planet. Sci.*, 34:37–45, 2006.
5. John Michalakes and Manish Vachharajani. Gpu acceleration of numerical weather prediction. *Parallel Processing Letters*, 18(04):531–548, 2008.
6. Anthony Nicholls and Barry Honig. A rapid finite difference algorithm, utilizing successive over-relaxation to solve the poisson–boltzmann equation. *Journal of computational chemistry*, 12(4):435–445, 1991.
7. Peter Pacheco. *Parallel programming with MPI*. Morgan Kaufmann, 1997.
8. Tom Ross and Neal Lott. A climatology of 1980-2003 extreme weather and climate events. 2003.
9. J Silcock and Andrzej Gosci´ nski.´ *Message passing, remote procedure calls and distributed shared memory as communication paradigms for distributed systems*. Citeseer, 1995.
10. WL Smith, WP Bishop, VF Dvorak, CM Hayden, JH McElroy, FR Mosher, VJ Oliver, JF Purdom, and DQ Wark. The meteorological satellite: overview of 25 years of operation. *Science*, 231(4737):455–462, 1986.
11. Wim Vanderbauwhede and Tetsuya Takemi. Twinned buffering: A simple and highly effective scheme for parallelization of successive over-relaxation on gpus and other accelerators. In *2015 International Conference on High Performance Computing & Simulation (HPCS)*, pages 436–443. IEEE, 2015.

1. Actual performance tested by mbw tool (apt install mbw). 2Disabled by command: swapoff -a [↑](#footnote-ref-1)
2. Installed by command: apt install openmpi-bin mpich libopenmpi-dev libopenmpi2 4https://github.com/etascale/argodsm [↑](#footnote-ref-2)
3. https://github.com/clarenous/argodsm-SOR#test [↑](#footnote-ref-3)
4. https://github.com/clarenous/argodsm-SOR#results [↑](#footnote-ref-4)