PARALLEL CONCORDANCE BENCHMARK

by

Malak Saleh Aljabri
in partial fulfilment of the requirements for the degree of
Master of Science
MSc Advanced Internet Applications

Supervised by :
Dr.Phil W. Trinder

School of Mathematical and Computer Science

Heriot Watt University

August 2010
ABSTRACT

Multi and many core architectures are dominating general purpose computer platforms today, enabling multiple tasks to be computed faster than on single-core machines. However, to take full advantage of multi-cores, applications must be parallelized. A range of alternative parallel programming models are available, and common parallel technologies are OpenMP and MPI: the de facto standards for shared memory and distributed memory parallel programming models.

This dissertation presents the construction of a new parallel benchmark, a concordance program, and presents a comparative evaluation implementations in two common paradigms (MPI and OpenMP) on a multi-core architecture. The concordance program is designed to analyze a text and to locate the occurrence or occurrences of each word and sequence of words, and list the pages on which these sequences occur.

Versions of the concordance benchmark have been developed in the C and Haskell programming languages. The concordance program reads a text file with page breaks, and a sequence length (N), and constructs a concordance listing the pages where the words, or sequences of words including those across page boundaries, appear. The key data structure used in both the C and Haskell programs is a hash table, where the key is the word or the sequence
of words, the value is the frequency. Due to the high abstraction overhead of Haskell, the C version is found to be 3 times faster whilst the Haskell code is more than 3 times shorter.

Two parallel algorithms were designed, in MPI and openMP, and problems such as finding sequences across worker boundaries and load balancing are addressed. The naive parallel versions are profiled and benchmarked on multicore architecture using different profiling tools for each version. Several means to reduce overheads are investigated to develop tuned parallel versions of the concordance.

A performance comparison of the OpenMP and MPI concordance programs on an 8 core architectures shows that OpenMP achieves a greater performance, with a speedup of 5.3, than MPI, with the speedup of 2.6 as a result of the high communication cost required for the MPI version. The effort required to tune the OpenMP version is less than that required for MPI, as a direct benefit of the shared memory model. On the other hand, MPI delivers shorter code than OpenMP. The resulting code size from performance tuning of the OpenMP version is over 10000 lines, 90% of which is for reducing the synchronization overhead by dividing the hash tables to over than 600 hash tables, whereas the resulting code size for MPI is 571 lines.
ACKNOWLEDGMENTS

First, all praises and thanks be to Allah, the Almighty, for helping me finish my graduate studies and this research.

Professionally, I would like to offer my appreciation and gratitude to my supervisor, Dr. Phil Trinder, for his brilliance and amazing insight. His support, encouragement, and patience helped me during my study. I am grateful for his helpful advice and guidance during the creation of this dissertation.

I would also like to thank Dr. Greg Michaelson, for suggesting the topic of this project and his support and willingness to help at any time.

My sincere thanks also go to Mr. Mustafa Aswad and Mr. Khri Armih, for their constant willingness to answer my questions and to all my friends at Edinburgh and at Saudi Arabia for their encouragements and support.

Personally, special thanks and love to my parents, brothers, and sisters, for their steadfast support during my stay in the UK. The continual encouragement and prayers of my parents have been extremely instrumental in motivating me to pursue the studies that led to this achievement.

Words cannot even begin to express the debt of gratitude and admiration I feel for my husband Naif and my little son Abdulaziz for their support and sacrifice during my study. Their impact on me has been greater than I could say. I am truly grateful.
# Table of Contents

1 Introduction .................................................. 1
  1.1 Context .................................................. 1
  1.2 Aims and Objectives ....................................... 2
  1.3 Contributions ........................................... 3
  1.4 Dissertation Structure .................................. 4

2 Literature Survey ............................................. 5
  2.1 Parallelism ................................................ 5
    2.1.1 Introduction .......................................... 5
    2.1.2 Parallel Architectures ................................. 6
    2.1.3 Parallel Architectures Trends ......................... 12
    2.1.4 Parallel Programming ................................. 14
      2.1.4.1 Parallel Programming Approaches ................ 14
      2.1.4.2 Levels of Parallel Technologies .................. 14
      2.1.4.3 Parallel Programming Models ..................... 15
    2.1.5 Parallel Languages ................................... 15
      2.1.5.1 MPI ............................................. 16
      2.1.5.2 OpenMP .......................................... 18
    2.1.6 Performance Analysis and Benchmarking ............ 20
  2.2 Concordance ............................................... 21
    2.2.1 Concordance Problem .................................. 21
    2.2.2 Parallel Concordance ................................. 22

3 Sequential Design, Implementation, and Performance .......... 23
  3.1 Sequential Design .......................................... 23
    3.1.1 Choosing the data structure ......................... 23
    3.1.2 Writing an Efficient Algorithm ...................... 24
  3.2 Implementation ........................................... 28
    3.2.1 C Implementation ..................................... 28
    3.2.2 Haskell Implementation ............................... 28
  3.3 Performance Comparison ................................... 28

4 Parallel Design and implementation .......................... 32
  4.1 Parallel Design ........................................... 32
4.1.1 Load Balancing .............................................. 33
4.1.2 Finding sequences across worker boundaries .......... 34
4.1.3 Calculating the page numbers .............................. 35
4.2 OpenMP Parallel Algorithm .................................... 35
4.3 MPI Parallel Algorithm ........................................ 37
4.4 Parallel Implementation ........................................ 39
  4.4.1 OpenMP Implementation ................................. 40
  4.4.2 MPI implementation ...................................... 40

5 Experimental Results .............................................. 42
  5.1 Experimental Methodology .................................... 42
  5.2 OpenMP Performance ......................................... 43
    5.2.1 OpenMP Preliminary Results ........................... 43
    5.2.2 OpenMP Profiling Results ............................. 45
    5.2.3 OpenMP Performance Tuning ............................ 46
      5.2.3.1 Dividing the Hash Table .......................... 46
      5.2.3.2 Increasing the Computation Size ................ 47
    5.2.4 OpenMP Final Results ................................ 47
  5.3 MPI Performance ............................................. 49
    5.3.1 MPI Preliminary Results .............................. 49
    5.3.2 MPI Profiling Results ................................. 51
    5.3.3 MPI Performance Tuning ................................ 52
      5.3.3.1 Converting Blocking Communication to Non Blocking Communication ................. 52
      5.3.3.2 Compressing the Messages ........................ 53
      5.3.3.3 Communication Through Intermediate Masters .... 53
      5.3.3.4 Change the MPI Implementation .................. 53
      5.3.3.5 Increasing the Computation Size ................. 55
    5.3.4 MPI Final Results ................................... 55
  5.4 Parallel Concordance Comparison ............................ 57
    5.4.1 Programming Model Comparison ........................ 57
    5.4.2 Performance Comparison ............................... 60
    5.4.3 Summary ............................................. 63

6 Conclusion ....................................................... 64
  6.1 Summary ................................................... 64
  6.2 Limitations ................................................ 65
  6.3 Future Work ................................................. 66

Bibliography ....................................................... 67

A Project Plan ..................................................... 75
# List of Tables

3.1 Length of code for Haskell and C sequential versions ............................... 29  
3.2 The Measured Runtimes for the sequential Haskell and C versions with  
Different Sequence Length ........................................................................... 29  
3.3 The Measured Runtime for the sequential Haskell and C versions with  
Different File sizes ....................................................................................... 30  

5.1 Runtime for Naive OpenMP for E1 ................................................................. 44  
5.2 Reducing Synchronization Overhead by Dividing the Hash Table ............. 47  
5.3 Runtimes for Tuned OpenMP for E2 .............................................................. 47  
5.4 Final OpenMP Runtime, Absolute Speedups, and Efficiency for E1 and  
E2 ..................................................................................................................... 48  
5.5 Runtimes for Naive MPI for E1 ...................................................................... 50  
5.6 Runtimes for MPI version for E1 and E2 on MPICH1 and MPICH2 ...... 55  
5.7 Final MPI Runtime on MPICH2, Speedup, and Efficiency for E1 and E2 56  
5.8 Code Size for all versions of the concordance program ............................ 60  
5.9 Final runtimes, speedups, and efficiency for MPI and OpenMP for E2 61
Chapter 1

Introduction

Section 1.1 explains the research context in parallel computing and multi-core architectures in which the project is developed. Section 1.2 gives the aims and objectives. Section 1.3 explains the contribution the research represents for computer science and academia. Section 1.4 outlines the structure of the dissertation.

1.1 Context

Due to the great advances in multi-core technology, the number of cores used has grown enormously and continues to grow, presenting important challenges for programmers and the computing industry in general. Multiple cores provide greater computing power within the same space. The changes in the complexity of hardware architecture required to do this mean that these processors are more challenging to use efficiently than previous designs [19]. The increasing use of multi-core has created a real need for parallelism [70].

Despite the great amount of research that has been conducted, there is no general agreement on how to handle multiple or multi-core CPUs [20]. Parallel computing is the way to exploit the full potential of multi-core architecture. When the program is well written, it performs much more quickly than a sequential program, allowing programs to carry out concurrent executions of multiple tasks by distributing the work among the available cores on the same processor [19]. Existing sequential applications generally need to be re-programmed in order to work in parallel [20] and while this is sometimes relatively straightforward, updating and testing codes for multi-core can be quite complex, requiring considerable time and effort [19].

When writing or converting applications, programmers should take into account that the new timing dynamics between the cores can give rise to new kinds of programming errors because if cores are waiting for each other to complete a task, they may freeze or become unsynchronized (known as race conditions) [19]. Software meth-
ods and tools are now available for programmers to make the most of parallelism, and are already being used to great effect in High Performance Computing [20].

OpenMP and MPI, as different parallel programming approaches, are notable among them. The decision of which one to choose will depend on the particular application and hardware.

OpenMP uses only shared memory. It was developed as a response to cumbersome native thread programming. It uses threads but the details are not visible to the programmer. Its higher level of abstraction makes threads easier to handle but, while it is easy to code, it lacks flexibility.

MPI is frequently used in High Performance Computing. It works by passing memory-copying messages from one processor to another, and is particularly effective when the processors do not share the same local memory. It is also suitable for use with multi-core, because programs ported to MPI do not need to be re-programmed for use on multi-core [20].

Having different parallel programming models increases the need to have an up-to-date evaluation of the available options for programming on multi-core machines. The aim of this project is to construct a parallel concordance benchmark using MPI and OpenMP as different parallel programming models, and to make a comparative evaluation between them. Concordances are used in text analysis to locate words and sequences of words and their frequency, and list the pages where they occur. They provide ease of access to information and are an extremely valuable research tool. To create a concordance is an extremely lengthy and laborious process. By using parallel programming, a concordance can be produced much more quickly, thoroughly and effectively, thereby contributing significantly to the world of research.

1.2 Aims and Objectives

The principal goal of the project is to develop a new parallel benchmark, a parallel concordance program, and make comparative evaluations on multi-core architecture.

The project aims to develop a new parallel benchmark, a parallel concordance, as follows: Given a text file of N words with page breaks, to construct a concordance listing the pages containing:
1. each word occurrence in a text of n words, and
2. each word sequence occurrence in a text of n words.
The project also aims to find solutions to a number of interesting problems such as: finding sequences across page boundaries, load balancing, and analyzing and reducing different kinds of overhead.

These benchmarks will be constructed using two different parallel languages: C with MPI, and C with OpenMP. The concordance versions will then be benchmarked on multi-core architecture, and a comparison between the different parallel programming languages will be carried out in terms of programming effort, length of code, speedup, and efficiency.

1.3 Contributions

The steps taken to carry out this project, and presented in this dissertation, are as follows:

- A literature survey has been undertaken of concordances, their history and importance for text analysis, and parallelism, covering parallel architectures, parallel architecture trends, parallel programming approaches, models and technologies: (Chapter 2).

- Two sequential concordance versions in the C and Haskell programming languages have been developed. Each version reads a text file with page breaks and the length of sequence (N), and constructs a concordance listing the pages on which each word or sequence of words occurs, and finds sequences across page boundaries. To the best of my knowledge, this is the first time an attempt has been made to construct a parallel concordance benchmark with sequences longer than 1. The key data structure used in both the C and Haskell programs is hash table, where the key is the word or the sequence of words, the value is the frequency, and a linked list represents the page numbers on which they occur. The C version is approximately three times faster than the Haskell version, due to the overhead of the high level language. The Haskell code, however, is more than three times shorter, for the same reason: (Chapter 3).

- Two parallel algorithms have been designed for the concordance benchmark in MPI and OpenMP, and solutions found to some interesting problems, such as finding sequences across worker boundaries and load balancing: (Chapter 4).

- The naive parallel versions are profiled and benchmarked on multi-core architecture using different profiling tools for each version. Different ways of reducing different kinds of overheads are investigated to develop tuned parallel versions of the concordance: (Chapter 5).
A comparison of parallel programming languages has been made in terms of: programming effort, length of code, speedup, and efficiency: (Chapter 5).

A performance comparison has been undertaken of the OpenMP and MPI concordance programs on an 8 core architectures and shows:

- The OpenMP version achieves a speedup of 5.3 while the mpi version achieved a lesser speedup of 2.6 as a result of higher communication costs.
- In order to reduce the synchronization overheads by 92%, the tuned OpenMP version has over 600 hash tables.
- Several alternatives have been investigated to improve the MPI performance including compressing data for communication, replacing blocking to non-blocking communication, and grouping data for communication: (Chapter 5).

1.4 Dissertation Structure

The content of this dissertation is structured as follows:
chapter 2: Literature Survey of Concordance and Parallelism.
chapter 4: Parallel Design and implementation.
chapter 5: Experimental Results includes Comparison of the Performance and Programming Models
chapter 6: Conclusions includes a discussion of the limitations and future work.
Chapter 2

Literature Survey

This chapter presents a survey of literature on parallelism and concordance. Section 2.1: outlines relevant literature on parallelism, including parallel architectures, trends, parallel approaches, levels of parallelism, and models (mpi and OpenMP), performance analysis and benchmarking. Section 2.2: discusses the problems of compiling concordances, tracing their history from the Middle Ages, to the beginning of the computer era, ending with parallel concordance.

2.1 Parallelism

2.1.1 Introduction

The aim of parallel computing is to increase an application’s performance by executing the application on multiple processors, thereby increasing the speed of the execution. Using a set of processors that work cooperatively means that we can concentrate our computational resources processors, memory, or I/O bandwidth - on a given problem and achieve our aims much more efficiently and more quickly [100]. The other great advantage offered by parallel computing is that it deals with problems that are too large or too complex to be handled by a single computer, especially computers that have a limited memory. A single computer resource can only do one thing at a time, with the sequence of steps running in an arbitrary order [93]. With multiple computing resources, on the other hand, different tasks can be divided among a number of processors, which then execute the tasks simultaneously. In addition to this, non-local compute resources can be used on a wide area network. All this increases not only the speed, but also the scope of what can be achieved dramatically.

In all, parallel computing makes much of our life run more smoothly, allows us to carry out extremely complex operations that would not have been possible with traditional sequential computing, and is extremely efficient in terms of time, performance and money.
However, parallel computing has a number of negative aspects to parallel computing. In practice, it requires a huge amount of highly complex work on the part of the programmer. At present, there is no proper coding for the processor to tell it how to distribute tasks correctly. Creating a parallel algorithm requires a number of steps that are not required for a sequential programme, such as task decomposition, mapping, and handling communication/synchronization issues and as yet there are insufficient tools to deal with this completely. Furthermore, these additional steps mean that parallel programming is more prone to error. Traditional techniques for reviewing and correcting defects and for improving performance are not applicable to parallel computing.

Portability is another difficulty in parallel computing that needs to be considered. In order to move a parallel software system from one parallel machine to another, a lot of work will be needed which might include rebuilding the whole system [93]. That is why parallel computing is generally considered hard for programmers, and needs programmers with advanced computer programming skills. Some systems appear now to make things easier, such as high-level parallel programming technologies like OpenMP and GPH.

### 2.1.2 Parallel Architectures

All parallel computers are made up of three different building blocks: the processors, the memory modules, and an interconnection network. Over the last decades there has been a gradual development in the degree of sophistication of each of these building blocks, but what makes one parallel computer different from another is the way they are arranged. The parallel computing processors themselves are now basically the same as those used in single processor systems [93].

Parallel computers have been classified in different ways, but probably the most widely used classification is the one proposed by Michael J Flynn in 1966, [48]which is known as Flynn’s Taxonomy. Flynn classified computing architectures into four categories based on the number of instruction streams and data streams available:

1. **Single-instruction, single Data stream (SISD)**
   This category has little or no concurrency compared with other categories. A computer with a single processor would be an example of this category. This kind of computer has only one CPU executing a single instruction stream on a single data stream. Nevertheless, modern computers of this kind are capable of some concurrency of execution. Examples of this are superscalar architecture and pipelining. The first of these, superscalar architecture, supports the dynamic identification and selection of multiple independent operations that may be executed simultaneously [82]. With Pipelining, the various circuits in the
CPU are divided into functional units. These are set up in a pipeline, which can produce a result during each instruction cycle [77].

2. **Single Instruction Multiple Data stream (SIMD)**

This category covers computers where a single instruction stream is performed on multiple data streams. There are two main examples of machines in this category. The first example is an array processor machine, in which many processor elements are performing the same operation on different data at the same time. The second example is a vector processor, which consists of a single control processor and many subordinate processors. At the beginning of each instruction cycle, the instruction is broadcast to all subordinate processors by the control processor. According to Flynn [48,93] this achieves concurrency of processing instead of concurrency of execution. The disadvantage of this kind of machine is that it consists of only one control processor. This means that many of the processors will be idle for long stretches of time when there are conditional branches. These machines are also relatively expensive in terms of distributed-memory MIMD systems [77].

![Figure 2.1 Single Instruction Multiple Data Architecture](image)

3. **Multiple-Instruction, Single-Data stream (MISD)**

This kind of architecture is not very common. In fact some experts do not place any machine in this category. Some experts say that it is only mentioned for the sake of completeness [72]. According to the description given of Flynn's taxonomy by Ben Motter and Jason Furente, [28] the category basically consists of multiple processors operating redundantly, which would seem to defeat the purpose of parallelism. However, they point out that there are significant uses for MISD, such as its applications for the Space Shuttle, where redundancy is critical, because if one processor fails, there would still be multiple identical processors to handle the tasks of the mission.
4. **Multiple Instruction Multiple Data (MIMD)**
Computers in this category consist of multiple processors. So, they have different processors that perform different instruction streams on different data streams. Each of these processors is able to work independently by having its own CPU, control unit, and ALU. This is the main difference between MIMD and SIMD. There are a number of related factors that have promoted development of MIMD architectures. The first is that computers in this category offer a high level of parallelism. Second, MIMD architectures are much faster, and are able to handle computing problems of much greater complexity than sequential architectures, which means that it is no longer necessary to depend on developing further refinements for pipelined vector computers for complex scientific applications. Third, MIMD is extremely cost effective and this has undoubtedly been a great incentive for the promotion of this kind of architecture [44].

![Figure 2.2 Multiple Instruction Multiple Data Architecture](image)

Computers with MIMD architecture are divided into two categories, based on the organization of memory: Shared Memory and Distributed Memory. According to Skillicorn and Talia [93], the main difference between the two is that the interconnection network, in the first (shared-memory MIMD), connects all processors to the memory, so that different processors can access the same variables. In the second (distributed-memory MIMD), each processor has its own local memory and accesses values in other memories by using the network. This means that different parts, or sub-tasks, of a computational task are distributed to multiple processors, each with its own memory space, and then the results from each processor are reassembled into one solution.

**Shared Memory**

In this model, there is a single shared memory which is available for all processors to access via a direct interconnection network. Communication among processors can
be achieved through a reading or writing to the shared memory and occurs implicitly as a result of conventional memory access instructions.

Shared memory systems can be classified as: SMP (Symmetric Multi-Processor systems), which, as this name implies, means that all processors share memory and IO equally and can access the same memory location at the same speed, and NUMA (Non-Uniform Memory Access) where the memory is closer to some processors and therefore can access some memory locations faster than others. [27]

Interconnection Networks
Interconnection networks in shared memory architectures can be divided into three main types: bus, cross-bar and multi-stage.

1. Bus-based architecture
   Basically means that wires inside the computer act like a bus route carrying data from one part of the computer to another. This is the simplest kind of interconnection network. The problem with this kind of architecture is that all the CPUs share the same communication path to all of the memory. If multiple processors are attempting to access memory at the same time, the bus can become saturated, which leads to long delays between the instructions to be performed. A bus has a limited bandwidth and, therefore, bus-based architectures are not able to accommodate a large number of processors [44, 77]. An example of this architecture is a SMP.

2. cross-bar switch based architectures
   In this architecture, there are multiple pathways between the different CPUs, and the CPUs and the memory. Because these paths are not shared, and there are more of them, the system does not suffer from the saturation problems which is found in bus-architectures and, therefore, the response time is quicker. In this kind of architecture, a crossbar switch of $n^2$ cross-points is used to connect n
processors to n memories [44]. A crossbar is essentially a rectangular mesh of wires with switches at the intersection points, and terminals on the left and at the top. However, in order to have more than eight processors in a single machine, two or more crossbars must be connected in a ring, which will need a corresponding increase in the number of hardware switches. The more switches there are, the more the cost of the machine increases. When a processor accesses memory attached to another crossbar, the access times will be slower, which is not the desired aim, but this is a performance-cost compromise that is adopted by most designers of machines of this kind. Architectures of this kind are Non-Uniform Memory Access, or NUMA [44,77].

3. Multi-stage Interconnection Networks (MIN)

They offer a compromise between the price/performance alternatives offered by crossbars and buses. An MXN MIN connects n processor to n memories by developing multiple stages, or banks of switches, in the inter-connection pathway. [27] Switching in each stage is governed by one bit in the address of the memory unit being accessed. The address bits of a memory unit select a path between the CPU and the memory unit it wishes to access. Multiple CPU-to-memory can occur concurrently as long as their paths in MIN do not intersect. An example of a computer which has a multi-stage interconnection network is the BBN Butterfly, which has a NUMA architecture [64].

It is relatively easier to implement parallel applications on shared memory architecture than distributed memory architectures. However, there are some problems to be solved such as cache coherency and data access synchronization.

**Cache Coherence**

One of the main problems to be found with all shared memory architectures that allow the caching of shared variables is cache coherence. When a processor wants to access a shared variable in its cache, as this variable has been accessible to other processors, there is no way for the processor to know if the data it is accessing has in fact been modified in any way by another processor. Snooping protocols are usually used to determine when data in the shared memory has been changed [82] [44].

**Data Access Synchronization**

When processors are cooperating to perform a computation, they may need to be synchronized in various ways. In shared memory computers, only one process can be engaged in at a specific time. To prevent, for example, a processor trying to access data before another processor has completed an updating task on the same data, a synchronization mechanism should be used. One kind of synchronization is known as *barrier synchronization* which ensures that no process can proceed beyond a specific
point, or barrier, until every process has reached the same barrier [82] [44]

Distributed Memory

In distributed-memory MIMD machines, each processor has its own individual memory location. This means that each processor has no direct knowledge about the memory of the other processors in the machine. Data must be passed from one processor to another as a message in order to be shared.

This kind of architecture requires the programmer to be involved in specifying the distribution of data among processors, and explicitly specifying how data is to be passed from one processor to another through the interconnection network.

The main features of this kind of architecture are: scalability, which is the ability to increase the number of processors in a machine, and high performance, which makes this kind of architecture suitable for complex scientific applications.

![Generic Distributed Memory Architecture](image)

**Figure 2.4** Generic Distributed Memory Architecture

There are two main classes of distributed memory architectures: *MPP* (Massively Parallel Processors) and *Clusters*.

With MPP, the processors and the network infrastructure are specifically designed to work closely together in a given parallel computer. These systems are extremely scalable and can support thousands of processors in one system.

Clusters, on the other hand, are made up of ready-made computers connected by a ready-made network. Systems of this kind are becoming more widespread and more powerful due to improvements in network technology. They are much less expensive than MPPs and therefore more accessible to organizations who want to install parallel computing [72].

**Interconnection Networks**

There are a number of different kinds of interconnection network: ring topology architecture, mesh topology architecture, and tree topology architecture.

1. **ring topology architectures**

   The communication diameter can be reduced by adding chordal connections,
which can increase fault tolerance. Fixed-size message packets that include a node destination field are used. This type of topology is most suitable for a small number of processors executing algorithms that are not dominated by data communication [44].

2. **mesh topology Architecture**
   A two-dimensional mesh has $n^2$ nodes, each of which is connected to the four nodes closest to it. Communication can be increased by adding diagonal links, or by using buses to connect the nodes by rows or columns [44].

3. **Tree topology architectures**
   They are constructed to support divide-and-conquer algorithms for searching and sorting, image processing [44].

### 2.1.3 Parallel Architectures Trends

The power of the conventional CPU has reached the point where the processor’s performance and speed are limited and can no longer cope with the rapid changes taking place in recent applications. A recent trend in high-performance computing depends heavily on exploiting parallelism.

**Multi-Core**

There is a gradual move towards integrating multiple processing cores within a single processor [85] [90]. Cores consist of a die of two individual processors. Each processor die contains many processing elements sharing a common L2 cache. The whole die works as traditional multiprocessing architecture except for the fact they are sharing the L2 cache in multi-core platforms [79].

There are advantages gained from the design of multi-core architecture: first, using multi-core leads to high level of parallelism in a program and, thereby increases performance and speed. Second, since many processing elements are sharing the same L2 cache, the communication overhead is reduced dramatically between these processing elements.

Several research studies have been conducted to examine the interconnections in multi-core architecture and its impact on the design of the future multi-core system [39]. Other studies have examined the impact of multi-core architecture in cluster computing [34].

In response to the recent increase in using multimedia and game applications, special purpose processors, called ”accelerators”, have been. In these accelerators, a combination of fine-grain and coarse-grain parallelism is exploited by using heterogeneous architectures of tens or hundreds processing cores [35]. These accelerators are
mainly used to achieve higher parallel code performance by speeding up the parts of
the applications which are compute-intensive [88] [35].

Different kinds of these accelerators have been developed, such as GPUs (graphics
processing unit), FPGAs (Field programmable gate arrays), and cell BES (Cell
Broadband Engines). Using these accelerators leads to significant increase in an
application’s performance, and high efficiency in terms of energy and money. Both
GPUs and FPGAs, along with other accelerators, can process work which the CPU off-
loads, and then send the results back when processed [88].

We will now look at examples of these accelerators in more detail: GPGPU is used for
computing on graphics processing units. It consists of many powerful and
flexible many-core processors with high memory bandwidth. These accelerators have
been successfully exploited to increase the performance of wide range of applications
including scientific, imaging and database applications. GPUs have become increas-
ingly important as powerful computing resources, not only for graphics, computer
games or films, but also in science, engineering, and financial modelling. They are
more energy efficient than CPU architecture [12].

FPGAs are non-conventional processors consisting of a matrix of programmable fine-grain
cells. These logic cells can be programmed by the user to fit their needs, and
connected into circuit of arbitrary complexity. During the last decade, the role of
FPGAs in embedded systems has become increasingly important as accelerators for
CPU-based protocols. They are extremely flexible and use cutting-edge technology.
They are widely used in security systems such as video surveillance. [4] [43]

These accelerators have massive parallel computing resources, which, together
with the fact that they are becoming increasingly easier to program, makes them
extremely suitable to use for accelerating the compute-intensive and, particularly,
the data parallel parts of applications [88]. Most accelerators today are available as
add-in boards, but in the future they will undoubtedly be incorporated in the com-
puter systems, probably located on-chip with the CPU, thereby reducing overhead
communication. GPUs and video processors will be especially prevalent [88].

From Multi-Core to Many Core
In 1965, Gordon Moore observed that, since the integrated circuit was invented, the
number of transistors per square inch had doubled, and that it would continue to do
so in the foreseeable future. His prediction has largely been correct, and has become
known as Moores Law. However, around 2007, Fred Pollack, of Intel, observed that
"performance increase is roughly proportional to [the] square root of [the] increase in
complexity”. What this means is that when the logic in a processor core doubles -
following Moores Law - it will only deliver 40% more performance. In other words,
it is a law of diminishing returns. With Multi-core systems becoming more widely adopted, and the business world and private individuals becoming more concerned about running costs and energy efficiency, Pollacks rule has become increasingly more relevant as a complement to Moores Law [71].

Multi-core architecture has provided an enormous improvement on performance and power efficiency, but, with ever-increasing technology of scaling, and a slower rate of increase in performance due to sub-threshold leakage current and the slowing down of supply voltage scaling, the consumption of power will become unreasonable in order to perform the required tasks. Multi-core architecture, therefore, will no longer meet all our needs. What is needed is optimal compute performance with reasonable power consumption. The answer lies in many-core systems. With this kind of architecture, each small core delivers lower performance than a large, complex core, but the total compute throughput possibility is much greater than a multi-core system with the same die size and same power consumption. Hundreds or thousands of small cores deliver far greater compute performance than has previously been possible, with affordable expenditure of energy. This kind of architecture marks a turning point in computer sciences, building on past research and, at the same time, opening up possibilities for new, qualitative advances, which will be particularly significant in the field of commodity hardware. [29,71]

2.1.4 Parallel Programming

2.1.4.1 Parallel Programming Approaches

There are two main approaches to parallel programming: implicit parallelism and explicit parallelism [45,69]

1- Implicit parallelism: This approach, which is also called auto parallelization or parallel compiler [56], keeps users away from the coordination aspects of parallelism. All they have to do is to develop a sequential version of their application and it will be automatically parallelized by the compiler. However, this higher level parallelism is extremely hard to achieve. Although there are several high level programming languages like GPH, the programmer still has to do some work specifying the coordination aspects, and there is no implicit parallelism.

2- Explicit parallelism: This approach involves the programmer in explicit specification of most of the parallelization, depending on the level of the parallel programming languages being used. Parallelization involves several tasks, such as decomposition of tasks, mapping tasks to processing elements, managing communication of data between processing elements.

2.1.4.2 Levels of Parallel Technologies

Since there is no pure implicit parallelization approach, several parallel programming languages and libraries are used. They are classified into three levels depending on
the parallelization effort needed from the programmer: low, mid and high. **1- Low level:** Low level parallel programming languages require the programmer to specify all the details of parallelization. An example is Java Socket, which is a very low level interface for specifying the inter-process communication details [68]. It has been used widely to provide distributed applications which are efficient and flexible. However, it involves writing long and complex details [40].

**2- Mid level:** Languages and libraries of this level encapsulate the common parallel code in a parallel library that can be used efficiently with several mid level computation languages like Fortran, C and C++ [45]. An example of this approach is the MPI library, which has been widely known and used to support parallel programming.

**3- High level:** The main purpose of high level parallel libraries and languages is to provide as much implicit parallelism in the tasks as possible. Examples of this include GpH, OpenMP.

---

**2.1.4.3 Parallel Programming Models**

Parallel programming models are abstractions of a computer system that enable you to express ideas in a particular way. The following are the most common kinds of models in use, all of which can be used with different types of hardware [18] [58].

- **Shared Address Programming:** With this model, information is stored at shared locations in order to communicate with different cooperating processing elements. Several mechanisms can be used to control access to the shared memory.

- **Message Passing:** Message passing is the most commonly used parallel programming model with distributed memory architecture. There is no shared location that can be accessed by all the processing elements. Instead, information is sent from a specific sender to a specific receiver. In this model, each processing element is performing the same operation but on different data. This is called **SPMD** Single Program multiple data programming models.

- **Data parallel Processing:** In this model, the cooperation is more strictly organized. Different elements of a data set are acted on by different processing elements at the same time. Following this, the information is exchanged globally and then the processing elements continue working together. A global reorganization of data can be carried out by accessing to shared addresses [18].

**2.1.5 Parallel Languages**

This section summarizes briefly some important issues related to MPI as a mid-level parallel programming model, OpenMP as an example of high-level parallel programming model.
2.1.5.1 MPI

In the 1990s, a committee of vendors, government labs and universities worked together to produce a standard specification for developers and users of message passing library. The result was Message Passing Interface. MPI is a set of library routines defined for C/C++ and FORTRAN. Since it is a standard interface, code written for one system can easily be ported to another system within those libraries. MPI is the only message-passing library that can be considered standard, and lends itself to virtually all distributed-memory programming models. It provides the programmer with a portable, flexible and efficient parallel application. However, according to Tony Hey [57], MPI might become less useful in the future because of the effect of two trends:

1- The increasing use of the web and web service protocols, which makes it possible to build more efficient software for distributed systems
2- A movement towards Multi-Core processors due to the difficulty of increasing clock speed as feature size becomes ever smaller.

Parallelizing an application using MPI is not an easy process. The programmer must be involved in each aspect of the coordination such as specifying how the work is to be divided among the processing elements, balancing the work to be done, managing the communications of the processors.

Moreover, there is no way in MPI to separate the communication and the computation parts of the code, as there is with other high level parallel programming languages such as GPH. MPI does not allow parallelizing the program incrementally as OpenMP does.

When an MPI program starts, several processes of the program will be created and run by a specified number of processors. Since there is no single shared memory, these processors need to communicate with each other in order to exchange data. The way these processors communicate is by sending and receiving messages. The communication can be carried out over a high speed network which connects the processors together [81]. At the simplest level, an MPI program uses a master process that sends off work to worker processes, which receive the data, perform tasks on it, then send the results back to the master process which combines all the results [77].

When the message is sent from one sender to one receiver, the communication mode is called one_to_one communication. The MPI sending request requires several piece of information: the sending process, the receiving process, the address in the memory of the sent data, a message identifier, and the group of processors which are going to receive the message (communicator) by using the MPI_SEND() and MPI_RECV() calls.

With point to point communication, it is also necessary to specify whether the sending and receiving calls are blocking or non-blocking. By blocking send or receive, the call will suspend the program execution until the buffer of the message being sent or
received is safe to use. A non-blocking call will not suspend the program execution. Instead, it will start the communication and return immediately. [9]

Another communication mode is collective communication, which involves all the processes in a communicator. MPI supports several different collective communications such as broadcast, where the same data is broadcast by a single process to all other processes in the communicator, and reduce in which all processes in the communicator contain an operand which is combined by successively applying a binary operator such as sum. A distributed data structure is collected into single process using gather, whereas scatter does the opposite job by distributing the data structure across the processes [82].

According to Brent Wilson [100], several applications can be developed using six functions: MPI_Init(), MPI_Comm_rank(), MPI_Comm_size(), MPI_Send(), MPI_Recv(), MPI_Finalize(). However, according to Sergei Gorlatch [54], using send and receive is considered harmful because it results in complicated programming process and therefore leads to a parallel program that is harder to understand and difficult to predict. They strongly suggested that send and receive statements be avoided as much as possible and replaced with the collective operations which are much more efficient in terms of simplicity, programmability, expressiveness, performance and predictability.

**MPI Example:**

This example shows how to use MPI to parallelize a loop which sums a 100 element integer array using 4 processors. The array is divided so that each processor will sum 25 elements of the array [81].
Listing 2.1 MPI Example

```c
#include <mpi.h>

Main(int argc, char ** argv)
{
    int i, p, id ,sum, result ,intArray[100],myChunk[25];
    MPI_Status status;
    /* Initialize MPI. */
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    /* initializes the array to the numbers 1..100 */
    for(i=1;i<100;i++)
        intArray[i] = i ;
    /* perform different actions depending on whether this is the master processor or worker.*/
    if (id == 0)
    {
        /* If this is the master , then distribute the work among the workers. */
        for (i=1; i<p; i++)
        {
            /* Copy master processor's data into its chunk array */
            MPI_Send(&intArray[i*25], 25, MPI_INT, i, 100, MPI_COMM_WORLD);
            /* copyArray copies n integers from one array to another */
        } /* copyArray[intArray, myChunk, 25]:} */
        /* Now that the problem is distributed, solve it. Each processor will have it's share of the work in myChunk array. */
        sum = 0;
        for (i=0;i<25;i++)
        {
            sum = sum + myChunk[i];
        } /* If this is the master , then distribute the work among the workers. */
        if (id == 0)
        {
            /* Send the results back to the master processor. */
            /* I receive each workers results and compute the total result. */
            for (i=1;i<p;i++)
            {
                /* I receive each workers results and compute the total result. */
                MPI_Recv(&result, 1, MPI_INT, i, 200, MPI_COMM_WORLD, &status);
                sum = sum + result;
            } /* I receive each workers results and compute the total result. */
        } else
        {
            /* Do something with the final result. */
            MPI_Send(&sum, 1, MPI_INT, 0, 200, MPI_COMM_WORLD);
        } /* Do something with the final result. */
        MPI_Finalize(); /* MPI shut down. */
    } else
    {
        MPI_Send(&sum, 1, MPI_INT, 0, 200, MPI_COMM_WORLD);
        MPI_Finalize(); /* MPI shut down. */
    }
}
```

2.1.5.2 OpenMP

OpenMP is a de facto standard application programming Interface (API) used mainly with shared memory architecture to provide parallel applications. It is not a new programming language, rather, it is a specification that can be added to some programming languages such as Fortran, C and C++ to specify the coordination aspects of a parallel program [46] [81].

OpenMP consists of a set of compiler directives, supporting library routines and environment variables to specify the parallelism, and program runtime characteristics [81]. The compiler directive is used to inform the compiler which region of the code should be implemented in parallel. To accomplish this, the programmer marks the region to be executed in parallel by inserting a parallel directive at the beginning of the region, and inserts an end parallel directive at the end. When the program starts execution, it is referred to as a process. This process will be allocated a memory space. Since OpenMP uses a thread paradigm, the process will begin with one thread which works as a master. When the master reaches the parallelized region, a team of workers will
be created and continue working together. All these threads are sharing the same memory and therefore it is not necessary for them to send or receive messages between them. Implicit barrier synchronization mechanism is used, so no thread will be able to progress beyond the end of parallelized region until all other threads in team reach the same point. After that, all threads in the team will terminate, and the pre-existing master will continue executing the program until it encounters another parallel directive. Multilevel parallelism is also possible. In this case, each thread in a team can create another team of threads and become its master.

This model of parallel programming is called the fork join model. The master thread will work sequentially until it encounters parallel directives, then it will fork off a team of new threads. This team then works in parallel until the end of the parallelized part, and then joins together. So the master thread can continue working as before until it encounters another parallel directive, and so on. A number of threads created within a parallel region can be specified by the programmer by setting the environment as variable : OMP_NUM_THREADS or using a call to the library : open_set_num_threads.

It is worth mentioning that each created thread within the parallel region has its own stack which is used for subordinates invoked by that thread [46]and, therefore, each thread is able to work without the possibility of interfering with other thread’s stack frames.

OpenMP has been successfully used to develop high performance parallel applications due to several advantages. First, it is simple to learn and to use with little programming effort. Moreover, it provides high-performance applications that are able to be run on different shared memory platforms and by different numbers of threads.

Furthermore , as a result of being a directive-based approach , the same code can be developed on a single processor as well as multiprocessor platforms, where in the former the directives are simply considered as comments and therefore will be ignored by the compiler and successfully implemented sequentially [46] [27] Another important advantage of OpenMP is that it allows parallelization to be carried out incrementally. By starting from a sequential program, the programmer needs simply to add the directives which express the parallelism. [80] [87] . Although OpenMP is considered a high level parallel programming model, the parallelization task is not always easy and straightforward. The programmer still needs to think carefully of how to exploit parallelism efficiently. Some tools have been developed to facilitate the parallelization processes and thereby significantly reduce the programming efforts. Examples of these tools include SGIs developer environment, which is designed specially for making the processes of parallelization, debugging and tuning much easier [11]POST project , which is an “EU-funded project which aims to reduce the human effort involved in the creation of OpenMP code“ [24]. Another example is KI/KAP/pro tool set which facilitate the development of OpenMP applications for
There are several issues related to parallel programming using OpenMP need to be considered by the programmer such as, Load balancing, scheduling and Race condition.

**Load balancing**
Load balancing is a common problem in OpenMP, as any multi-processing system. It is important to ensure that the work is distributed evenly across all the threads. This is vital, because it ensures that all the processors are busy almost all the time, thereby avoiding any processor resources remaining idle.

**Scheduling**
In OpenMP, the load balancing is often due to problems with thread scheduling. The programmer must specify how the work is to be shared out across the threads, otherwise, each processor will redundantly execute all of the code. There are a number of ways in which OpenMP can handle thread scheduling to prevent inefficiency. Among these are: static scheduling, dynamic scheduling or guided scheduling.

**Race condition**
Sometimes, the use of threads can create a problem known as race condition. This arises when more than one thread can modify the same variable or variables simultaneously. If this happens, highly erroneous or unreliable results will be produced creating chaotic situations.

**OpenMP Example:**
This example shows how to use OpenMP to parallelize a loop which sums a 100 element integer array.

```
#include <omp.h>

main () {
  int i, intArray[100], sum = 0;
  /* initializes the array to the numbers 1..100 */
  for(i=1;i<=100;i++)
    intArray[i] = i;
  //parallel region ..
  #pragma omp parallel shared(intArray, sum) private(i)
  {
    #pragma parallel for reduction(+: sum) /* Make the for loop a parallel region */
    for (i=0;i<100;i++) {
      sum = sum + intArray[i];
    }
  } /* end of parallel region */
}
```

### 2.1.6 Performance Analysis and Benchmarking

An important step in deciding whether or not our parallel program is achieving the desired level of efficiency is by measuring its performance. This allows the program-
mer to estimate the amount of parallelism which can be exploited and predict how much improvement can be achieved as a result of adding more processors, as well as knowing what improvement to expect in the speed of execution. The general formula for measuring the speedup is given by: \( S = \frac{T_1}{T_2} \), where \( T_1 \) is the time of executing the program sequentially, which means implementing the program using one processor. \( T_2 \) denotes the time of executing the program in a parallel way, using multiple processing elements. In other words, the speedup is ratio of sequential execution time and parallel execution time [82].

The increased parallel speedup achieved is due to exploiting an efficient amount of parallelism and therefore indicates the success of parallelization [27]. According to Michael J. Quinn [82], there are three kinds of operations that are implemented by parallel algorithms:

1- Computations which have to be implemented sequentially.

2- Computations which are able to be parallelized.

3- Coordination operations.

Obviously, all parallel programs contain different numbers of these operations. By increasing the processing elements, the execution time of the parallelized operations will be reduced, but the execution time for the sequential operation will remain the same for any number of processing elements. Eventually, the execution time will be completely determined by the execution time of the sequential operations [82] [27]. This is what Amdahl was describing in his law, which puts an upper limit on the speedup that can be achieved. The formula for Amdahls Law is: \( S = \frac{1}{f_{par}/P+(1-f_{par})} \) [27]. \( f_{par} \) refers to the operations that are performed in parallel, \( P \) is the number of processors. The ideal case would be when \( f_{par} = 1 \), that is when the whole code is executed in parallel and therefore, the expected speedup is equal to the number of processors - \( S = P \). The perfect achieved speedup is linear and is ideally achieved by increasing the number of processors.

It is important to mention that there are a number of obstacles to achieving the linear speedup, such as overhead communication, which occurs when the number of processing elements become more than the work to be done. So, at the same point, the speedup will start to decline as the number of processors increases. The efficiency of the parallel program can also be measured by dividing the speedup by the number of processors, which is computed by the following formula: Efficiency = \( \frac{\text{SequentialExecutionTime}}{\text{Processors used} \cdot \text{ParallelExecutionTime}} \) [82].

### 2.2 Concordance

#### 2.2.1 Concordance Problem

A concordance is an alphabetical list of the main words used in a text, with the preceding and following words that provide the context. It is also a gathering of different texts which are linked in some way by the words or ideas they share [73].
In general terms, concordances are frequently used in linguistics for: comparing different usages of the same word, analysing key words and word frequencies, finding and analysing words and phrases, creating indexes and word lists. A dictionary will give you the definition of words in a text, or words used by a particular author, but a concordance gives us a deeper understanding of meanings and how those words are used. The more text you use to create the concordance, the more information it will give you about the nature of the language used in a particular corpus [99].

Concordances are among the simplest information retrieval tools. They originated from methods that go back to the middle ages. Originally, because of the amount of time involved, concordances were only carried out on works of special importance. The first recorded concordance was in Latin. It was of the vulgate Bible. They were lists of words arranged alphabetically to help scholars find a piece of text or references to something in particular [73]. It was compiled by the Dominican friar Hugh of St Cher (d.1262) with the help of 500 other monks. Over the centuries, other scholarly concordances were carried out on the Bible, Milton’s Paradise Lost and the works of Shakespeare, to give some famous examples [73].

Until the computer era, they were elaborated by painstaking and highly skilled scholars and took many years to complete. Computers have made it possible to create complex concordances in a fraction of the time. The first computer-generated concordance of the works of Shakespeare, was conducted by Martin Spevack in 1968 [89].

In Computing, concordance can be used to analyse a text in a given language, developing tools for searching and to choose appropriate vocabulary. Moreover, it can be used for developing effective databases and for Spelling and Grammar checks. An important example is the concordance of the Holy Quran in English which is meant for people who want to understand Quran and have no command for Arabic Language [61].

2.2.2 Parallel Concordance

Constructing a Concordance is an important problem in text analysis. The main purpose of my project is to construct a parallel concordance. Experiments on parallelizing different applications in different areas of computing have been carried out, such as pattern matching, image processing, and bio-informatics [38]. To the best of my knowledge, no prior work has been done to parallelize concordance problems, except an MPI program to count the occurrence of each word in a text, which has been implemented by Agarwal [83].
Chapter 3
Sequential Design,
Implementation, and Performance

This chapter presents the design, implementation and performance of the sequential concordance program. Section 3.1 discusses the sequential design. Section 3.2 discusses the implementation of the sequential version in C and Haskell programming languages. Section 3.3 discusses and compares the performance of each version in terms of speed and scalability.

3.1 Sequential Design

Designing the sequential versions of the concordance program requires two important aspects which should be correctly specified in order to develop an efficient program [92]. These are: choosing the data structure and writing an efficient algorithm.

3.1.1 Choosing the data structure

There are a number of different ways of organizing data, e.g. linked lists, hash table, tree, stacks and queues.

Since the purpose of the program is to read the text file and to construct a concordance of each sequence, as well as to search for the pages where the sequence occurs, an efficient data structure is needed that will be able to maintain a large amount of data as well as to facilitate searching and updating information. To meet these specifications, three alternative data structures can be used: Linked List, Binary Tree and Hash table.

Linked List is simple to implement and is a widely used data structure. It uses memory efficiently. However, since the program will deal with large files of data, and data will be accessed frequently, searching with a linked list is slow, especially when the data is not sorted with the complexity O(n). Moreover, Linked lists do not support random access [67, 91].
Binary Tree is a good choice for many purposes, and might be used since it is more efficient than linked lists, but it is not the best choice for our needs since the searching complexity is $O(\log n)$.

The best structure for organizing the concordance data, as it supports the most efficient type of searching, is hash table [92]. Hash table provides constant time searches, no matter how big the data is, which is $O(1)$. Hash table is not ideal to use with some problems, because it has a number of drawbacks. However, since insertion and searching are the main operations needed in our program, and these operations are really fast with hash table, it is the best choice for this project. Single linked list can be used easily to store the pages numbers since it is not known in advance how many times each sequence occurs and on which pages. The following figure shows the representation of the hash table, which consists of three things: the key which is the sequence, the value which is the frequency of use, and the list of pages on which the sequence occurs.

![Figure 3.1 Hash Table Representation](image)

3.1.2 Writing an Efficient Algorithm

A good algorithm should have three desirable properties: correctness, efficiency and ease of implementation [92]. To demonstrate the correctness of the algorithm, we first need to give a careful description of the problem and also a specification of the set of allowed instances and the required properties of the output.

The problem:
Given a text file of $N$ words with page breaks, construct a concordance listing the
pages on which: every word occurs, and every sequence of words from length 1 to N-1 occurs.

**Input:**
The length of the sequence: N, and the file name.

**Output**
Concordance listing the pages as shown in Figure (3.2):

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Frequency</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2383</td>
<td>2, 3, 5, 6, 7,...</td>
</tr>
<tr>
<td>A AN</td>
<td>1</td>
<td>244</td>
</tr>
<tr>
<td>A AN ANGEL</td>
<td>1</td>
<td>244</td>
</tr>
<tr>
<td>ALLAH</td>
<td>2988</td>
<td>10, 11, 12,...</td>
</tr>
</tbody>
</table>

Figure 3.2 Example of the Program Output

**The properties of the input:**

- The file should be a text file with no size limitation.
- It may include some special symbols, commas, and numbers which should be excluded from the sequence.
- The length of the sequence should be a positive integer number.

**The required properties of the output:**

- The output should include only sequences of words.
- The output should print all possible sequences less than or equal to N.
- The program should print the sequence, the number of occurrences, and also the pages where the sequence occurs.

An important design decision I have made was to implement one parallel concordance benchmark instead of two as they both are constructing a concordance and using the same algorithm with the deference of the sequence length. This can be achieved by letting N be controlled by a parameter that can be changed easily. So, when setting the value of N to 1, it will be equal to the first program which will construct a concordance listing the page on which every word occurs. When setting the value of N to larger than 1, it will be equal to the second program, which will construct a concordance listing the pages on which every sequence of words from length 1 to N-1 occurs.
Sequential Algorithm

In the sequential version, the program is designed to run on one processor and to be computationally cheap. An interesting problem encountered in designing the sequential algorithm is how to find sequences across page boundaries. This is solved simply by treating the page marker as a space and increasing the page numbers. Figure (3.3) shows the algorithm proposed to develop the sequential version of the concordance program.

**Figure 3.3 Sequential Concordance Construction Algorithm**

Activity Diagram:
Figure 3.4 Activity Diagram for the Concordance Construction Algorithm
3.2 Implementation

Since the main purpose of my project is to compare different languages used to solve concordance problems, the data structure used must be the same. The full C and Haskell implementations can be found as files: "sequential_c_version.c" and "sequential_haskell_version.hs" in the sequential directory on the attached CD.

3.2.1 C Implementation

The Hash table is implemented in C version using Glib [21]. GLib is a general-purpose utility library, released under the GNU Library General Public License (GNU LGPL), providing a large number of useful data types, macros, type conversions, string utilities, file utilities, a main loop abstraction, and so on. It works on many UNIX-like platforms, Windows, OS/2 and BeOS. It is a utility library for C that provides the programmer with the several facilities. For pages, single linked list is implemented, which will create a new node each time a sequence is found on a new page. The insertion to list is done from the head of the list, so the page numbers are sorted by default in a descending order.

3.2.2 Haskell Implementation

There are a number of issues relating to problems existing in the hash table data structure supported by Haskell [23]. These problems include the performance as well as the low speed due to the fact that the garbage collector traverses the entire array needlessly’ [16].

A benchmark posted by Jon Harrop [2], demonstrated that the implementation of hash table in Haskell is extremely slow. Don Stewart et al [76] write about this problem. They said that a well-implemented, purely functional tree data structure can perform more efficiently than a hash table, and that should not mean that your code will perform less well. Fortunately, these problems have been fixed in the latest GHC version 6.12.1 [14] as stated by Simon Marlow, which is used to develop the sequential Haskell version.

3.3 Performance Comparison

In this section, a comparative evaluation of the sequential versions in C and Haskell is presented for the parallel concordance benchmark.

Haskell is a functional programming language in which a program is written as a set of functions definitions and other values [96]. The main characteristics of Haskell which make it different from other languages are: it is a polymorphically-typed, lazy,
purely-functional language and support referential transparency. C on the other hand and most other programming languages like Java, pascal etc. are imperative languages. A program in these languages consists of sequence of commands required to be executed sequentially one after another.

The C program establishes the specific steps that must be taken by the machine in order to perform a given algorithm, and for the most part, the code is dealing with low-level details of data manipulation, whereas in the Haskell program, the algorithm is encoded at a much higher level, and is both shorter and clearer [7].

Since Haskell is high level functional language, where a lot of low level implementation details are hidden under the abstractions, it leads to a short code which is much easier to develop and understand, and also increases the programmer’s productivity. For the concordance benchmark, all the sequential programs have been measured on a common multicore architecture, double core machine, comprising Intel(R)core(TM)2Duo processor P8700 running at 2.53 GHz and 4GB RAM running under Ubuntu release 10. The Haskell program was more than three times shorter than its C counterpart as shown in table (3.1). The user friendliness of Haskell, however, comes with a price, because the performance of Haskell applications is slower than that of their C counterparts. This is due to the overhead resulting from the high abstractions. For the concordance programs as shown in table (3.2) the C version was about three times faster than the Haskell version for a file size 1mb and sequences length up to 8.

Table 3.1 Length of code for Haskell and C sequential versions

<table>
<thead>
<tr>
<th>Programming Language</th>
<th>C</th>
<th>Haskell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line of Code</td>
<td>256</td>
<td>74</td>
</tr>
</tbody>
</table>

Table 3.2 The Measured Runtimes for the sequential Haskell and C versions with Different Sequence Length

<table>
<thead>
<tr>
<th>Runtimes</th>
<th>C</th>
<th>Haskell</th>
<th>C/Haskell Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence Length (N)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0.056</td>
<td>0.84</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>0.19</td>
<td>1.86</td>
<td>9.7</td>
</tr>
<tr>
<td>4</td>
<td>0.79</td>
<td>4.20</td>
<td>5.3</td>
</tr>
<tr>
<td>6</td>
<td>1.66</td>
<td>6.52</td>
<td>3.9</td>
</tr>
<tr>
<td>8</td>
<td>2.47</td>
<td>10.82</td>
<td>4.3</td>
</tr>
</tbody>
</table>

Figure (3.5) depicts the measured runtime of the sequential Haskell in comparison to the C programs for a file size of 1 MB and increasing the number of N.
For a fixed sequence length and increasing data size as shown in table (3.2), the C/Haskell ratio decreases to around four times slower. Table (3.3) and Figure (3.6) depict the measured runtime of the sequential Haskell in comparison to the C programs for a fixed sequence length of 1 word and increasing the file size up to 20 MB.

**Table 3.3** The Measured Runtime for the sequential Haskell and C versions with Different File sizes

<table>
<thead>
<tr>
<th>File Size(MB)</th>
<th>Runtime (second)</th>
<th>C</th>
<th>Haskell</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.05</td>
<td>0.80</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.10</td>
<td>1.62</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.21</td>
<td>3.42</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0.31</td>
<td>4.94</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0.42</td>
<td>6.64</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0.50</td>
<td>7.38</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>0.64</td>
<td>10.22</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>0.72</td>
<td>11.68</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0.83</td>
<td>14.04</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>1.03</td>
<td>15.22</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>1.12</td>
<td>16.70</td>
<td></td>
</tr>
</tbody>
</table>

Figure(3.6) shows that the hash table in Haskell has expected performance: doubling the size of the input doubles the execution time. In other words, the linear time is as predicted.
As a final conclusion, when performance is paramount for an application, or when a low-level algorithm needs detailed tuning, it may well be advisable to use an imperative language such as C rather than Haskell, because the way the computation is carried out can be more closely controlled [16]. On the other hand, when the speed of application is not critical, and the longer execution time is acceptable, it is preferable to use Haskell in order to benefit from the short development time.
Chapter 4

Parallel Design and implementation

This chapter presents the design and implementation of the parallel concordance program. Section 4.1 discusses parallel design and presents the parallel algorithm for OpenMP and MPI and presents the solutions to some related issues. Section 4.2 discusses the implementation of the parallel versions in C with OpenMP and C with MPI.

4.1 Parallel Design

The design of a parallel algorithm focuses on the most intensive computation, aiming to parallelize it, in this case, constructing the concordance. Other computations can be further parallelized, which involves sorting the hash table as well as the pages linked list, but since there are several algorithms for sorting and searching in parallel [82], and the main concern in this research is to construct a parallel concordance benchmark, we are focusing on parallelizing the work of constructing the concordance. In designing the parallel versions of the concordance benchmark, a SPMD model is used, as it is the most suitable structuring pattern, provides flexibility, and is widely used in large-scale scientific applications [72, 84](Section 2.2.4.3). In SPMD, computations are assigned to workers using their ID, and parallelism is exploited by a data decomposition. Even though SPMD is highly scalable, it requires a great effort to program in terms of coordination between the different workers, such as work and data distributions among workers, which needs to be explicitly expressed. For MPI, this style is strongly favoured and is closely related with the message passing environment [72]. On the other hand, OpenMP is widely used for fine grain loop-level parallelism since it supports incremental development as well as being easy to implement. However, it is also possible to use SPMD with OpenMP, paying the price of
programming effort needed for coordination aspects. One of the most important reasons for using SPMD with a message passing paradigm in a lot of existing applications, is the difficulty of achieving high performance with OpenMP when using loop-level parallelism [66]. A number of experimental results have demonstrated the excellent performance for different applications given by using SPMD with OpenMP [65,66]. The proposed OpenMP and MPI algorithms provide solutions for three important problems: finding sequences across worker boundaries (section 4.1.1), load balancing (section 4.1.2), and calculating page numbers (section 4.1.3).

4.1.1 Load Balancing

Load balancing is achieved by dividing the text into subtexts and assigning these subtexts to the workers. The size of each subtext is computed by dividing the file size by the number of workers and, since the remainder of the division is not large, the last worker is assigned the rest of the text, if any. Due to the cost of reading the complete file with the master and assigning subtext to each worker, each worker receives the offset and the size of the file to read, because the file is stored inside the same shared memory. Achieving optimal load balancing is subject to the granularity of the data [22,77]. Coarse-grained parallelism often results in small numbers of larger chunks, which makes load balancing more difficult. Moreover, the most effective factor for load balancing is the irregularity in the data, e.g. some workers may have a part of the text which includes more frequently used English words than other workers, and therefore will have more work to do.

The top 25 most commonly used words in the English texts (the, of, and, to, in, is, you, that, it, he, was, for, on, are, as, with, his, they, I, at, be, this, have, from) account for around a third of all printed material in the English language [3]. They are all form, or function words (articles, prepositions, conjunctions, plus some pronouns and auxiliary verbs), that is, words that are used to construct sentences and link the content, or information words together to express different relationships between them. [3,13].

The top 100 (as before + or, on, had, by, word, but, not, what, all, were, we, when, your, can, said, there, use, an, each, which, she, do, how, there, if, etc.) still have a high proportion of form words, but are beginning to include some content words verbs, nouns, adjectives and account for half of all written English texts [3,5]. If we look at the top 300 English words, which constitute around 65% of written texts, we see that they contain progressively more informational words verbs, nouns, adjectives, adverbs, numbers etc (including over, new, sound, take, only, little, work, know, place, year, live, back, give, most, very, high, need, food) [37,75]. It is clear, then, that some workers may have a larger number of these words in the assigned text than others, and therefore have more work to do.
4.1.2 Finding sequences across worker boundaries

For sequences of length 1, there will be no boundary problem since each worker has its own chunk of data and will construct concordance for its own chunk. The main problem is for sequences of a length more than 1. In this case, the data shared between the workers is limited by the boundaries of chunks. A good solution to this issue is achieved by surrounding the data structure for each chunk with a ghost boundary. The ghost boundary is used to include a replication of data at the boundaries of neighboring chunks. As a result, each worker will have its own chunk and zero or more copies of the ghost boundary [72]. Using ghost copies has two benefits [72]. First, it may significantly reduce the communication overhead by consolidating the number of messages into fewer messages of a larger size. Moreover it is possible to overlap the communication of ghost cells with the update of the interior of the arrays, as they are not dependent on the data within the ghost copy. In our case, the problem has been solved, following the same approach, by replicating data on boundaries. This means that each worker will have an extra N-1 words from the next worker’s data, with the exception of the last worker. Each worker will, therefore, generate sequences using N-1 words from the next worker’s data, and the next worker will read the same N-1 words but will use these words to generate sequences with the following words N, N+1, .. and so on. Figure (4.1) explains this solution with an example. This solution has been tested and the program found the sequences in ghost boundaries correctly.

![Diagram](image)

**Figure 4.1** Finding Sequences Across Workers Boundaries

It is also sometimes possible to have the worker boundary in the middle of the word.
In this case, the first worker will proceed by reading this word as if it is in its own chunk. When the next worker starts reading from the beginning of its boundary, which is here the middle of the word, the worker will read 1 more character before the boundary to be sure that the boundary is in the middle, and will then skip the rest of this word since it has already been considered by the previous worker.

4.1.3 Calculating the page numbers

Since each worker will work on its allocated chunk of data, there is no way for each worker to know the last page the previous worker reached, in order to continue counting from this number. There are two possible solutions to this problem:

1. The master reads the whole file and counts how many pages the file text contains. The master can, therefore, send to each worker, in addition to the offset and the size of the allotted data, the page number that the worker has to start counting from.

2. Assume a large integer number as a page boundary so that each worker will start counting the pages using this boundary and the worker ID. For example, let the page boundary be 1000 000, so, the first worker will start from 1, 2, 3, and so on, and the second worker will start from page boundary * ID which is 2000 000, 2000 001, 2000 002, and so on. Finally, when the master begins sorting the hash table, it can recalculate the page numbers at the same time.

Since reading the complete file using the master in advance is expensive, and we need to focus on constructing the concordance, the second solution will be used.

4.2 OpenMP Parallel Algorithm

OpenMP supports fork/join pattern (section 2.2.5.2), and in the concordance the master thread is reading the file size and the number of available threads, and computes the size of work for each thread. In the parallel region, the master thread will fork to the number of specified threads, and each will have the offset and the size of file to work on. Since the memory is shared, all threads have access to the same hash table, as it includes the concordance results which need a synchronization mechanism to control reading from or writing to it. At the end of the parallel region, the group of threads will be joined to one thread, which will proceed to the sorting of the hash table and printing the final results. Figure (4.2) shows the proposed OpenMP algorithm for the parallel concordance benchmark:

Figure (4.3) presents the details of the main_process function which is called by all workers in order to carry out the main task of constructing the concordance.
The implementation of these algorithms can be found in file: "OpenMP_version.c" in the directory parallel on the attached CD.
4.3 MPI Parallel Algorithm

MPI supports master/worker parallelism (section 2.2.5.1), and in the concordance the 
master reads the file size, the number of available workers, and computes the size of 
work for each worker.

At the beginning of the parallel work, the master process is responsible for computing 
the worker size and offset to indicate the part of text assigned to each worker, and 
then sends them to the group of specified workers. As the program runs on shared 
file system (NFS), there is no need to distribute the file among workers. Instead, each 
worker receives the offset and the size of file to work on, and read the file. Since it 
is a message passing programming model, each worker has its own hash table, which 
includes the concordance results for its corresponding chunk of data.

At the end of the parallel work, each worker will convert its hash table into a suit-
able format for communication, and send it to the master, which is responsible for 
de-converting the received hash tables and combining them into one final hash table. 
Finally, the master will sort and print the final results.

Because MPI follows the message passing model, the communication plays a sig-
nificant role in the performance of the MPI version. Finding optimum granularity 
for the data decomposition, can be achieved by experimenting with a range of chunk 
sizes [72]. The granularity should be controlled by parameters that can be changed 
easily [72]. For this reason, the MPI version is designed so that we can change the 
number of messages each worker will send to the master, which will contain the con-
verted hash table.

Figure (4.4) shows the proposed MPI Master algorithm for the parallel concor-
dance benchmark.

Figure (4.5) shows the proposed MPI worker algorithm for the parallel concordance 
benchmark. The next figure (4.6) presents the details of the main process function 
which is called by all workers in order to do the main task of constructing the con-
cordance. The implementation of these algorithms found in the program "mpi_version.c" 
in the directory "parallel" in the attached CD.
a- Read the maximum sequence length (N) and the file name 
b- Get the file size 
c- Get the number of workers 
d- For I = 1 to num_of_workers:
   - Compute worker_size
   - Compute the offset as: worker_ID * worker_size
   - Send worker_size
   - Send the offset
End_For

e- Compute for the worker the total number of chunks the master expects to receive as:
   num_of_chunks * num_of_workers

f- For I = 1 to total_chunks:
   - Receive the size of the packed hash table
   - Receive the packed hash table
   - Unpack the received hash table to the final hash table
End_for

g- Receive the last page number of each worker

h- Compute the correct last page number for each worker
   For I = 1 to num_of_workers
      Last_page[i] = last_page[i] % PAGE_BOUNDARY + last_page[i-1] - 1
   End_for

i- Recalculate the hash table pages and remove page duplications if any

j- Sort the hash table

k- Print the hash table

Figure 4.4 MPI Master Algorithm

a- Receive the worker_size 
b- Receive the offset 
c- Specify Worker type based on the worker_ID 
d- Call main_process function by sending: (file, worker_size, 
   worker_type, worker_ID, offset) which will find the sequences, pack
   the hash table and send the packed hash table to the master, and also
   return the last page number for each worker to the master.

Figure 4.5 MPI Worker Algorithm
4.4 Parallel Implementation

In the OpenMP benchmark, there is no dedicated master as the memory is shared. However, as the MPI benchmark has a distributed memory, it needs a dedicated master in order to share out the work among the different workers and combine the results.

Debugging is a notorious problem, and in our case, particularly so, given the large size of the program, (571 lines for MPI and 10964 lines for OpenMP) making the use of printfs inadequate as a debugging technique. Moreover, the runtime error messages are, more often than not, useless. This can, in part, be explained by the fact that C is weakly typed and allows unsafe use of pointers, or array indices. In addition, memory management is explicit and prone to errors.

All of these things demonstrate the importance of having an efficient debugging tool to aid in the discovery and correction of errors. Consequently, the ”Totalview” debugger
is used, which allows the debugging of one or many processes and/or threads, and provides total control over the program execution [15]. It allows you to reproduce and troubleshoot problems occurring in concurrent programs that take advantage of threads, OpenMP, MPI, or computational accelerators. Care should be taken not to introduce a deadlock while organizing communication. Furthermore, barrier synchronization is used for more accurate timing [72].

4.4.1 OpenMP Implementation

OpenMP is implemented in Single Program Multiple Data (SPMD) model by spawning the specified number of threads in the parallel region. Each thread uses its id value, which has private scope for the purpose of specifying the area of a text on which the thread has to work. This is entirely based on the OpenMP parallel directive, which encloses the parallel region where a set of spawn threads execute concurrently [46,98], leading to coarse-grain parallelism. No for Loop-directives are used since they are not suitable to use here.

Since OpenMP is a shared memory model, the hash table is shared among all threads to communicate with each other by reading from and writing to it. Consequently, access to the hash table needs to be explicitly organized among all threads in order to avoid conflicting access and incorrect data [27].

It is highly recommended to use the atomic directive for synchronization, as it uses hardware support to ensure that the shared memory is atomically updated, which results in higher performance than using the critical construct [25,27,94]. However, the atomic directive is not used since it has a set of restrictions over the critical directive. For example, it can only be applied to a critical section consisting of a single assignment statement with a special format, whilst the critical directive can be applied to a critical region with an arbitrary number of statements [27]. For the purpose of parallel concordance, we have mutual exclusion through a critical construct, which is used to protect the updating of the hash table by looking up, inserting and updating operation.

4.4.2 MPI implementation

The implementation of the MPI version in Single Program Multiple Data (SPMD) model is achieved by using the Master/worker parallel programming pattern. Blocking communications are used for sending and receiving the worker-size, the offset and the packed hash tables.

Since the hash table consists of more than one element which is not a suitable format for ordinary communication in MPI as it is stored in non-contiguous memory
locations, it has to be converted to a suitable format for communication.

To achieve this, MPI provides three different techniques, each with its own positive and negative features. These are: count parameter, derived data type and \texttt{MPI\_pack()} / \texttt{MPI\_unpack()} functions [77]. The count parameter requires the data to be stored in contiguous memory locations [77,78] and derived data type is too complicated to implement, and in addition, many of its implementations perform poorly [32,33]. For this reason, the third mechanism, which is the one used for the MPI version of the concordance, is the pack and unpack functions. In order to send the hash table, the \texttt{MPI\_pack()} function is responsible for packing the entire hash table to a buffer, and then sending the buffer. When the buffer is received, the master will unpack the buffer using the function \texttt{MPI\_unpack()} to the final hash table [77].
Chapter 5

Experimental Results

This chapter presents the experimental results and performance comparison of the MPI and OpenMP versions of the concordance programs. Section 5.1 describes the experimental setup. Section 5.2 discusses OpenMP performance, giving the performance of the naive OpenMP version, showing the results of profiling the naive OpenMP version. It also discusses the different kinds of overhead that hinder the achievement of high performance, discusses the performance tuning and the ways of reducing the overhead, and shows the final results of the tuned OpenMP version. Section 5.3 discusses MPI performance, showing the preliminary results and the results of profiling the naive MPI version, and discusses the different kinds of overhead that hinder the achievement of high performance. It discusses the performance tuning and the ways of reducing the overhead, and shows the final results of the tuned MPI version. Section 5.4 presents a comparative evaluation of the MPI and OpenMP versions on multi-core architecture for the parallel concordance benchmark, covering both performance and programming models. Section 5.5 presents a summary of the final results.

5.1 Experimental Methodology

For these experiments, the parallel algorithms in C programming language using OpenMP and MPI library were implemented. The programs have been measured on a common multi-core architecture, eight-core machine, comprising two Intel Xeon 5410 quad-core processors, running at 2.33 GHz, with a 1998 MHz front-side bus 6144 KB and 8GB RAM running under CentOS release 5.5. The compiler used is the gcc version 4.1.2.

In order to achieve reliable readings, the reported data represent the median of 3 executions for all experiments and the measurements were made on up to 8 cores. The reported speedup are absolute speedup that are calculated as $Sp = \frac{T(1)}{T(p)}$, where $T(1)$ is the run time of the sequential program and $T(p)$ is the runtime on $P$ processors. The efficiency is calculated as $Ep = \frac{T(1)}{(P \times T(P))}$. 
In addition, for profiling the benchmarks, ompP 0.7.1 profiling tool is used for profiling the OpenMP version and mpiP Version 3.2.1 profiling tool is used for profiling the MPI version.

Finally, for all the reported experiments, two samples of files were used with different sequence lengths, as follows:

1. The size of the first text file is about 18 MB and the sequences of up to 10 words.
2. The size of the second text file is about 1 MB and the sequences of up to 50 words. The value of N is chosen to be long in order to increase the computation. The tested files contain a translated copy of the "holy Quraan" and some "sunnah" books. These text files are included in the attached CD in the directory text files.

### 5.2 OpenMP Performance

This section describes the performance of the naive OpenMP version of the concordance benchmark. Timings, scalability, runtime profiles of the parallel region, and overhead analysis of this benchmark are reported. In addition, performance tuning as well as the reduction of different kinds of overhead are discussed. Finally, the final results of the tuned benchmark are presented.

In order to obtain accurate measurements of the time spent on the concordance construction part of the benchmark, the function: `omp_get_wtime()` is used.

#### 5.2.1 OpenMP Preliminary Results

It is clear that having one shared hash table, with the team of threads competing to read from or write to it, will lead to synchronization problems or to consistency problems, e.g. losing some entries. In order to avoid such problems, a mutual exclusion construct must be used to coordinate access to the hash table and to ensure that the hash table is always up to date, with no two threads updating its contents simultaneously [46].

In the first attempt to have a parallel OpenMP version, as explained in the previous chapter, protecting the hash table is done by synchronizing reading from or writing to the hash table using critical construct. Table 5.1 shows the runtime measured for the naive OpenMP implementation on a different number of cores.

Unsurprisingly, no improvement in the runtime was achieved, but rather, the runtime was increased as the number of cores increased over the sequential version. As shown in Figure 5.1, the performance of this OpenMP implementation is much worse than the sequential version. This behaviour is expected as a direct effect of the synchronization overhead, which means that when a thread executes within the
### Table 5.1 Runtime for Naive OpenMP for E1

<table>
<thead>
<tr>
<th>Number of Cores</th>
<th>Runtime (Seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>48.35</td>
</tr>
<tr>
<td>2</td>
<td>42.7</td>
</tr>
<tr>
<td>3</td>
<td>47</td>
</tr>
<tr>
<td>4</td>
<td>52.1</td>
</tr>
<tr>
<td>5</td>
<td>56.7</td>
</tr>
<tr>
<td>6</td>
<td>61.8</td>
</tr>
<tr>
<td>7</td>
<td>74.2</td>
</tr>
<tr>
<td>8</td>
<td>82.6</td>
</tr>
<tr>
<td>Sequential Runtime</td>
<td>39</td>
</tr>
</tbody>
</table>

#### Figure 5.1 Parallel Runtime for the Naive OpenMP for E1

Critical sections, all other threads will remain idle, waiting for the current thread to finish. As a result, no more parallelism is exploited and the code within the critical section is implemented sequentially.

Moreover, the critical construct is the most expensive synchronization construct supported by OpenMP [46, 63]. This time is not only the time that each thread has to wait before the critical region, but also the time for performing the synchronization construct [94].

Furthermore, the abstraction provided by the OpenMP directives hides most of the implementation details from the programmer such as creation and destruction of the team of threads. As a result, a significant effect on performance of many OpenMP applications is caused by the compiler and the runtime system [49].

Experiments studying the performance characteristics of OpenMP constructs show that another source of overhead in OpenMP applications results from using the `PARALLEL` directive, which has the largest overhead of all other OpenMP directives [49].
5.2.2 OpenMP Profiling Results

Improving the performance of the parallel code requires using a profiling tool to ascertain the runtime characteristics of the OpenMP code and provide the programmer with the analysis of the application’s overheads for each parallel region as well as for the whole program.

We analyse the overhead of OpenMP application using the ompP profiling tool. OmpP is a profiling tool designed to help the programmer to understand the scalability behaviour of the OpenMP applications on shared memory architecture, which is an important step in developing scientific applications [50]. The ompP plays a great role in discovering and analysing different kinds of overhead which limit the application’s scalability. It determines the execution times for all OpenMP directives. It also analyses the overhead for each parallel region separately as well as for the whole program, and generates a profiling report upon the completion of the program execution.

The ompP overhead analysis report shows four overhead categories. These are: Synchronization, Imbalance, Limited parallelism and Thread Management overhead [50–52].

According to the ompP overhead report presented in Figure 5.2, the synchronization overhead is the dominating overhead affecting the increase in the runtime of the naive OpenMP program, with 63.92%, which represents the amount of waiting time spent by all threads in order to enter the critical section.

---
<table>
<thead>
<tr>
<th>ompP Overhead Analysis Report</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total runtime (wallclock)</td>
</tr>
<tr>
<td>Number of parallel regions</td>
</tr>
<tr>
<td>Parallel coverage</td>
</tr>
</tbody>
</table>

Parallel regions sorted by wallclock time:

<table>
<thead>
<tr>
<th>Type</th>
<th>Location</th>
<th>Wallclock (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0003 PARALLEL</td>
<td>firstomp.c (115-139)</td>
<td>263.39 (97.02)</td>
</tr>
<tr>
<td>R0001 PARALLEL</td>
<td>firstomp.c (101-103)</td>
<td>0.19 (0.07)</td>
</tr>
<tr>
<td>SUM</td>
<td>263.58 (97.09)</td>
<td></td>
</tr>
</tbody>
</table>

Overheads wrt. each individual parallel region:

<table>
<thead>
<tr>
<th>Type</th>
<th>Wallclock (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0003</td>
<td>2107.11 (84.53)</td>
</tr>
<tr>
<td>R0001</td>
<td>1.56 (100.00)</td>
</tr>
</tbody>
</table>

Overheads wrt. whole program:

<table>
<thead>
<tr>
<th>Type</th>
<th>Wallclock (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0003</td>
<td>2107.11 (82.03)</td>
</tr>
<tr>
<td>R0001</td>
<td>1.56 (0.07)</td>
</tr>
</tbody>
</table>

Figure 5.2 ompP Overhead Analysis Report

45
The report shows that there is no limited parallelism overhead. It shows also that the imbalance overhead is about 11.8%, which is difficult to reduce at this stage as it is related to the amount of work each thread has. Since the text in the file is irregular, some threads have a portion of text which contains a higher proportion of frequently used English words than others. As result, they may spend more time getting access to the shared hash table related to those common words. This is clearly related to the text file and not to the load balancing mechanism. Using `nowait` clause could help to reduce the time spent on the implicit barrier synchronization at the end of parallel construct [50]. Unfortunately, `nowait` clause can only be applied for loop-level parallelism. The last kind of reported overhead is thread management overhead, about 6.28% of which is related to the implementation of the underlying runtime system [46].

5.2.3 OpenMP Performance Tuning

As shown in the overhead analysis report, the OpenMP program suffers from the synchronization overhead, which is the main reason for this poor performance. It is clear that the key improvement here is the reduction of the overhead associated with synchronizing the access to the hash table.

5.2.3.1 Dividing the Hash Table

One way of reducing the synchronization overhead is to divide the hash table into multiple hash tables, and use a different lock for each one. This approach can ensure that no two threads are updating the same part hash table of the hash table at the same time, while allowing them to update different hash tables at the same time and hence significantly reducing the waiting time [46, 86]. Consequently, as experimental results show, dividing the hash table results in up to 92% synchronization overhead reduction, as well as more than 7 times performance improvement of the OpenMP concordance benchmark.

These results are presented in the Table (5.2) which presents the number of hash tables used, the percentage of synchronization overhead involved and the speedup on 8 cores. However, the price to be paid for the outstanding performance is the programming effort needed in order to divide the hash table and coordinate access among the large number of hash tables. Moreover, it yields a code which is long, difficult to debug, and prone to errors.

These results prove that SPMD programming style can successfully achieve a high performance, but requires a great programming effort in order to explicitly manage the memory, coordinate the parallel work and reduce the synchronization overhead involved. This clearly runs counter to the purpose of developing OpenMP, which is for it to be easy to learn and to apply.
Table 5.2 Reducing Synchronization Overhead by Dividing the Hash Table

<table>
<thead>
<tr>
<th>Number of Hash tables</th>
<th>Synchronization Overhead</th>
<th>Speedup on 8 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>63%</td>
<td>0.58</td>
</tr>
<tr>
<td>8</td>
<td>27.65%</td>
<td>1.65</td>
</tr>
<tr>
<td>54</td>
<td>21.6%</td>
<td>3.3</td>
</tr>
<tr>
<td>364</td>
<td>12.47%</td>
<td>4.2</td>
</tr>
<tr>
<td>657</td>
<td>4.86%</td>
<td>4.38</td>
</tr>
</tbody>
</table>

5.2.3.2 Increasing the Computation Size

This section presents the performance of the tuned OpenMP version for the second experiment where the file size is 1MB and N = 50. This experiment shows a significant increase in the performance for the OpenMP benchmark compared with the first one. This is the result of having a large amount of computation in terms of reading words, parsing characters and forming sequences. Consequently, each worker will spend more time on computation than on updating the hash tables. Table(5.3) shows the measured runtime for experiment 2 on different number of cores.

Table 5.3 Runtimes for Tuned OpenMP for E2

<table>
<thead>
<tr>
<th>Number of Cores</th>
<th>Runtime (second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>54.16</td>
</tr>
<tr>
<td>2</td>
<td>28.8</td>
</tr>
<tr>
<td>3</td>
<td>20.1</td>
</tr>
<tr>
<td>4</td>
<td>15.7</td>
</tr>
<tr>
<td>5</td>
<td>12.9</td>
</tr>
<tr>
<td>6</td>
<td>11.1</td>
</tr>
<tr>
<td>7</td>
<td>9.8</td>
</tr>
<tr>
<td>8</td>
<td>8.7</td>
</tr>
<tr>
<td>Sequential Runtime</td>
<td>46.5</td>
</tr>
</tbody>
</table>

5.2.4 OpenMP Final Results

The final results show a great improvements in the performance of the OpenMP code, as the synchronization overheads have been reduced to 4.86%. The hash table has been divided into 657 hash tables. Approximately 90% of the code was for reducing the synchronization overhead. Since the second experiment shows a much greater improvement, it is presented as
the final OpenMP results. The second experiment was tested with file size = 1MB and N = 50. Table (5.4) summarizes the runtimes (R), speedups (S) and efficiency (E) measured for final OpenMP program for the two experiments on different number of cores.

**Table 5.4** Final OpenMP Runtime, Absolute Speedups, and Efficiency for E1 and E2

<table>
<thead>
<tr>
<th>Number of Cores</th>
<th>E1 (N=10, File size = 18MB)</th>
<th>E2 (N=50, File size = 1MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Cores</td>
<td>R(seconds)</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>51.3</td>
<td>0.7</td>
</tr>
<tr>
<td>2</td>
<td>30.3</td>
<td>1.2</td>
</tr>
<tr>
<td>3</td>
<td>22.2</td>
<td>1.7</td>
</tr>
<tr>
<td>4</td>
<td>18.3</td>
<td>2.1</td>
</tr>
<tr>
<td>5</td>
<td>15.7</td>
<td>2.4</td>
</tr>
<tr>
<td>6</td>
<td>13.9</td>
<td>2.8</td>
</tr>
<tr>
<td>7</td>
<td>12.7</td>
<td>3.0</td>
</tr>
<tr>
<td>8</td>
<td>11.7</td>
<td>3.3</td>
</tr>
<tr>
<td>Sequential Runtime</td>
<td>39</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure (5.3) depicts the measured runtime with the number of cores compared to the sequential version for for the final OpenMP implementation for Experiment 1 and Experiment 2.

![Figure 5.3 Parallel Runtime for the Tuned OpenMP Version for E1(left) and E2(right)](image)

Figure 5.3 Parallel Runtime for the Tuned OpenMP Version for E1(left) and E2(right)
Figures 5.4 depict the speedup for the final OpenMP implementation of Experiment 1 and Experiment 2.

![Figure 5.4 Parallel Speedup for the Tuned OpenMP Version for E1 and E2](image)

5.3 MPI Performance

In this section, the performance of the naive MPI version of the concordance benchmark on an eight-core machine is presented. Timings, scalability, runtime profiles of the parallel work, and statistical analyses of all MPI functions of the benchmark are reported. Performance tuning as well as different ways of reducing overhead are also discussed. Finally, the final results of the tuned MPI benchmark are presented. In order to obtain accurate measurements of the time spent on the concordance construction part of the benchmark, the function: \texttt{MPI Wtime()} was used.

5.3.1 MPI Preliminary Results

Due to the master/worker model used in MPI, it is not feasible to run the MPI program with single core, since the design of the MPI program implies having one master, which is responsible for sending work and combining results, and which is not involved in the computational work.

In the first attempt to have a parallel MPI version, as explained in the previous chapter, the master was sending the worker size and offset. Each worker then received the worker size and the offset and started working on its area of the text file. At the end of the parallel work, each worker would pack its own hash table and send it to the master as one message. When an attempt was made to run this MPI implementation on multi-core for the first experiment, with file size = 18 MB, and N=10, it would not work, since the hash table that each worker had to send was too large and exceeded the maximum message size that can be sent successfully.
As a result, we had to change the number of messages each worker sends. We kept trying with different values for the number of chunks in order to find the best granularity, and found out that the best performance was with 12 chunks. Unfortunately, the performance was disappointing, as can be seen in Table 5.5. We obtained a significant reduction in the run time as the number of cores increased, but the performance of this MPI implementation was much worse than the sequential version, which had a runtime of 39 seconds.

<table>
<thead>
<tr>
<th>Number of Cores</th>
<th>Runtime (second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>101.7</td>
</tr>
<tr>
<td>3</td>
<td>52.0</td>
</tr>
<tr>
<td>4</td>
<td>48.3</td>
</tr>
<tr>
<td>5</td>
<td>45.6</td>
</tr>
<tr>
<td>6</td>
<td>47.4</td>
</tr>
<tr>
<td>7</td>
<td>46.7</td>
</tr>
<tr>
<td>8</td>
<td>46.8</td>
</tr>
<tr>
<td>Sequential Runtime</td>
<td>39</td>
</tr>
</tbody>
</table>

Figure (5.5) shows the large gap between the performance of the MPI version and its sequential counterpart.

Several factors have a negative effect on the performance of MPI on shared memory architectures. Different kinds of non-data communication overheads within MPI implementation have been studied by Balaji et al [26]. Some of these might directly affect the performance of the MPI version of the concordance benchmark. For example:

- **MPI stack overhead**: MPI stack overhead increases proportionally with the message size as a direct result of the protocol switching [55]. There are three different protocols for message transferring: short, for less than 1 kb, eager, for up to 128 kb and rendezvous for larger messages. [26].
• **Tag and Source matching overhead**: Messages in MPI are classified according to their attached tags. When the message is received, a search is carried out to identify it from the queue of posted receive requests. Problems of scalability may occur if the queue grows too long. [26].

• **Unexpected messages Overhead**: When attempting to receive required messages, the receiver treats all messages that have previously been sent as unexpected and they are put in a queue in the MPI stack until they are dealt with. This process can involve copying of data that can result in added overhead [26].

• **Communication Protocol of the MPI Implementation**: With the MPICH1 implementation, Miao [74] et al returned a poor performance for MPI on shared memory architecture to the double memory copying involved as it increases overhead.

### 5.3.2 MPI Profiling Results

As with the OpenMP version, improving the performance of the parallel MPI code requires using an evaluating tool to evaluate the runtime characteristics of the MPI code, and provide the programmer with an analysis of the application’s performance for each MPI function as well as for the whole program.

For this reason, profiling of the MPI version was carried out using the mpiP profiling library, which is a lightweight profiling library designed specifically for MPI applications [97]. Using mpiP results in far less overhead and less data than tracing tools, as it limits itself to collecting task-local statistical data regarding MPI functions. The only communication it uses occurs while a report is being generated, which is generally at the end of an experiment, and when merging all the results of the different tasks into a sole output file.

The mpiP library is relatively simple to use, since it collects MPI information through the profiling layer. The different sections of the output report show information about the performance of the experiment and the application time in MPI. The report also presents identification of all MPI call-sites within the application.

The most important section of the mpiP report is the overview of the top twenty MPI call-sites that consume the most aggregate time in the application [97].

When profiling the MPI version for the first experiment, the profiling report indicated in Figure(5.6) shows that sending the backed hash table function is consuming about 67.35% of the application time. This indicates that the performance of the MPI version is affected by the communication overheads. The MPI version was implemented on a shared memory machine using MPICH1 implementation and, the MPICH1 implementation of MPI does not take full advantage of shared memory for
communication [74]. This is because, in order to communicate data, a double copying system is used to copy data to and from system buffer, and therefore, the message passing from the sender buffer to the receiver buffer obviously takes more time [74].

### 5.3.3 MPI Performance Tuning

Several attempts were made to optimize the performance of the MPI version on a shared memory machine for the parallel concordance benchmark, as follows:

#### 5.3.3.1 Converting Blocking Communication to Non Blocking Communication

In the first attempt to tune the MPI version to improve its performance, blocking communication was changed to non-blocking communication. `MPI_Send()` was changed to `MPI_Isend()`, and managing `MPI_Request()` was used. In addition, calling for `MPI_Wait()` was used to wait for the completion of the message receiving. Non-blocking communication allows part of the message latency to be hidden by overlapping the communication with the necessary computation [47]. However, the non-blocking program took longer due to the time needed to wait for message completion. The mpiP profiling report shows that `MPI_Wait()` function was consuming as much as 72.46% of the application time.

Balaji et al [26] explained that was because of the overhead that was added by the
managing $MPI\_Request()$, which is required to allow the non-blocking operation to be completed. Allocation, initialization and queuing / de-queuing are required for these requests, within the implementation of MPI for each non-blocking operation, leading to the increase in the overhead [56].

### 5.3.3.2 Compressing the Messages

To reduce the communication volume, the messages are compressed before they are sent to the master. For this purpose, the Zlib library was used [17]. The Zlib compression library offers in-memory compression and decompression functions, which include integrity checks for the uncompressed data. This means that each worker will pack the hash table, compress it and send it to the master. When receiving the message, the master will uncompress the hash table, unpack it and build the final hash table.

Unfortunately, no improvement was achieved by using compression, because the time used to compress the data outweighed the benefit achieved by reducing the message size, leading to an increase in the communication time, which was about 70% of the application’s time as reported by mpiP profiler.

### 5.3.3.3 Communication Through Intermediate Masters

This attempt to improve the MPI version aims to reduce the communication to the master. Since we have several workers competing to send their packed hash table to the master, that could account for the increase in sending time. As a result, in this attempt, we changed the way of sending the message through an intermediary master.

Figure (5.7) shows how the messages were being sent. Each worker sent the message to another worker, who acted as an intermediary master, finally forwarding the message to the master. According to the profiling results, a very small improvement of the sending time was achieved, but the communication time was still high, reaching about 56% of the application time.

### 5.3.3.4 Change the MPI Implementation

The previous unsuccessful attempts to optimize the performance of the MPI, demonstrated the problem of MPICH implementation for communication on shared memory architecture. The problem was reported by Miao et al [74] as an effect of double copying of data for communication. The MPICH implementation of MPI is the impediment to good performance, due to the fact that this implementation takes little account of the underlying architecture [74]. The level of achievable performance for MPI depends on the implementation [30].

Extensive research has been carried out to improve MPI communication on shared memory architecture:
An optimization of MPI communication on shared memory architecture can be achieved by replacing double data copying protocol with single data copying protocol. This new communication protocol was designed and developed by Miao et al [74] and was demonstrated to outperform MPICH communication protocol with lower latency.

A newer version of MPI implementation, namely MPICH2, was developed to solve several problems such as improving the communication on multicore architecture. MPICH2 is a high-performance and widely portable implementation of the Message Passing Interface (MPI) standard (both MPI-1 and MPI-2).

MPICH2 aims to:
1- provide an MPI implementation to support different computation and communication platforms, including multicore architecture.
2- offer the possibility of cutting-edge research in MPI by using an easy-to-extend modular framework for other derived implementations [10].

Butinas et al [30, 31] also developed and implemented a communication system called Nemesis as a communication subsystem for MPICH2, providing scalability, high performance shared memory and multi-network communication. It uses lock-free queues in shared memory, so that only one receiving queue is needed, avoiding increased overhead.

Large Message Interface (LMI) is defined and added to the middle layer of MPICH2 to support the different mechanism of transferring large messages. This improves communication in shared memory architecture. Details of Nemesis design and implementation can be found in [30, 31].
Different mechanism for data transfer on SMP systems have been presented and classified by Buntinas et al [30]. They also evaluate the performance of these different mechanism based on latency, bandwidth, and its suitability for different kinds of MPI messages.

The MPICH2 version 1.2.1p1 is installed and the MPI version of the concordance benchmark is tested by considering MPI process as core, which is the common way of implementing MPI applications on multi-core architectures [10]. Results on table (5.6) show a great improvement in the runtime(R) of our MPI version for experiment 1, and experiment 2 on 8 cores.

\subsection{Increasing the Computation Size}

The results of the experiment 2 in Table (5.6) show that the MPI performance increased as a result of increasing the computation to communication ratio increased. This second experiment showed an increase in the performance for the MPI benchmark compared with the first one, as a result of having a large amount of computation in terms of reading words, parsing characters and forming sequences. Consequently, each worker was spending more time on computation than on packing and sending the hash table.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
\textbf{Cores} & \textbf{E1 (N=10 , File size = 18MB)} & \textbf{E2 (N=50 , File size = 1MB)} \\
\hline
2 & 101.7 & 92.3 & 71.3 & 67.3 \\
3 & 52.0 & 45.7 & 36.2 & 33.52 \\
4 & 48.3 & 33.8 & 27.2 & 23.0 \\
5 & 45.6 & 29.8 & 26.6 & 18.8 \\
6 & 47.4 & 30.0 & 26.3 & 18.6 \\
7 & 46.7 & 30.6 & 25.8 & 18.3 \\
8 & 46.8 & 30.4 & 26.2 & 18.0 \\
Sequential Runtime & 39 & 39 & 46.5 & 46.5 \\
\hline
\end{tabular}
\caption{Runtimes for MPI version for E1 and E2 on MPICH1 and MPICH2}
\end{table}

\subsection{MPI Final Results}

The final results show a significant improvement in the performance of the MPI code implemented on MPICH2, with around 30% improvement in the runtime. This result demonstrated the improvement of the current implementation of MPICH2 for shared memory communication.
In the second experiment, where the computation to communication ratio was increased, the MPI performance improved, with the absolute speedup of 2.6 on 8 cores. Table (5.8) summarizes the runtimes (R), speedups (S) and efficiency (E) measured for final MPI program for the two experiments on different numbers of cores.

Table 5.7 Final MPI Runtime on MPICH2, Speedup, and Efficiency for E1 and E2

<table>
<thead>
<tr>
<th>Number of Cores</th>
<th>R(second)</th>
<th>S</th>
<th>E</th>
<th>R(second)</th>
<th>S</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>92.3</td>
<td>0.4</td>
<td>0.21</td>
<td>67.3</td>
<td>0.6</td>
<td>0.34</td>
</tr>
<tr>
<td>3</td>
<td>45.7</td>
<td>0.8</td>
<td>0.23</td>
<td>33.5</td>
<td>1.3</td>
<td>0.46</td>
</tr>
<tr>
<td>4</td>
<td>33.8</td>
<td>1.1</td>
<td>0.28</td>
<td>23.0</td>
<td>2.0</td>
<td>0.50</td>
</tr>
<tr>
<td>5</td>
<td>29.8</td>
<td>1.3</td>
<td>0.26</td>
<td>18.8</td>
<td>2.4</td>
<td>0.49</td>
</tr>
<tr>
<td>6</td>
<td>30.0</td>
<td>1.3</td>
<td>0.21</td>
<td>18.6</td>
<td>2.5</td>
<td>0.41</td>
</tr>
<tr>
<td>7</td>
<td>30.6</td>
<td>1.2</td>
<td>0.18</td>
<td>18.3</td>
<td>2.5</td>
<td>0.36</td>
</tr>
<tr>
<td>8</td>
<td>30.4</td>
<td>1.2</td>
<td>0.16</td>
<td>18.0</td>
<td>2.6</td>
<td>0.32</td>
</tr>
<tr>
<td>Sequential Runtime</td>
<td>39</td>
<td>-</td>
<td>-</td>
<td>46.5</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure (5.8) depicts the measured runtime with the number of cores compared to the sequential version for the final MPI implementation for Experiment 1 and Experiment 2.

Figure 5.8 Parallel Runtime for the Tuned MPI Version for E1 (left) and E2 (right)

Figure (5.9) depicts the speedup for the final MPI implementation of Experiment 1 and Experiment 2.
5.4 Parallel Concordance Comparison

This section presents a comparison between MPI and OpenMP as different parallel programming models, followed by a comparison between them on multi-core architecture. Finally, a comparative evaluation between them for a parallel concordance benchmark is presented.

5.4.1 Programming Model Comparison

MPI is a de facto standard library designed and widely used to develop parallel applications on distributed memory architecture: Section (2.2.5.1).

MPI is relatively straightforward for the programmer from a conceptual point of view, but complex programs can be built, with numerous concurrent activities, which can sometimes lead to a program that is hard to follow and debug and much prone to error.

Developing programs for MPI that must function across a cluster of processors requires a programmer to be involved in a lot more work than when programming with a single server [41].

Parallelism can be expressed in a number of different ways, but in High Performance Computing, most parallel programmers use Message Passing Interface (MPI) [41].

OpenMP is a de facto standard Application Programming Interface, used to provide parallel applications on shared memory architecture: Section (2.2.5.2).

Using an incremental approach by adding OpenMP pragmas to the sequential code results in using operating system threads, which are like separate subroutines. These run concurrently, sharing the same memory space. Most often, OpenMP is used with loops which are augmented with these OpenMP directives, which are then used by the compiler to thread the loop automatically.
The advantage of this type of approach is that it may be possible to leave the original sequential program untouched except for directives, as these are considered as comments for the sequential version. This offers the possibility of recompiling for a sequential version, ignoring the OpenMP directives [41, 42].

Due to the growth of multi-core processors, parallel programs are increasingly needed for all kinds of computers, from the largest multiple processor clusters, to the smallest laptop [42]. Coding for multi-core is essentially the same as parallel programming. When deciding whether to use OpenMP or MPI, the question a programmer has to ask is whether an existing codebase will scale, or whether it is necessary to rewrite the code so that it will be more scalable on the multi-core architecture [42].

It might seem logical that OpenMP would be the best choice for using with multi-core, because it is designed for shared memory. However, the scaling benefits of OpenMP or threads should be seriously weighed up before embarking on a lengthy rewriting of the code. It may well be that using an existing MPI code base that performs adequately on multi-core will be perfectly satisfactory for a given application [42].

MPI uses a software library to transmit data, or pass messages, from one process to another, each with its own memory space, where the messages are copied from. Message passing is actually memory copying, and MPI is designed so that the messages can be passed between different processes on the same server, or on different servers [41, 42].

The one big difference between OpenMP and MPI is the OS process space. OpenMP programs run as a single process and the parallelism is expressed as threads, whereas MPI starts one process per core using the mpirun np 8 ... command [42].

In OpenMP, communication occurs by means of shared memory. This means that threads share access to a memory location. Communication with MPI programs on SMP systems also occur using shared memory, but the messages are sent by them writing from private to shared memory [41, 42].

It might seem obvious that sharing memory locations would be more efficient than sending copies of memory locations to other processes, but that is not necessarily the case. With MPI, single processes have exclusive access to all their process memory. In some cases, it may be considered to be more efficient to copy data (or send a message) instead of synchronizing on shared memory. However, the OpenMP model has the advantage of the threads being able to share access to all memory in the process space.

In some cases, therefore, some programs may be much more efficient because there is no need to have the large overhead of copying memory [41, 42].
An important point to bear in mind is the scalability of the application. Increasing the number of processors improves performance up to a certain point, and then, creating more threads or adding new processes may in fact damage performance [42]. MPI can be used with both shared memory and distributed memory architecture, offering a great scalability. OpenMP, which only works with shared memory, also offers good scalability, but has some drawbacks compared to MPI. It requires a shared address space, and therefore, scalability to the number of CPUs within one SMP node [60]. The limited parallel scalability is generally considered to be the price that must be paid for the ease of programming with a shared memory model. Distributed memory programming model is generally considered to be more difficult, but the only way to achieve higher levels of parallel scalability [46].

Distributed memory programming implementation APIs such as MPI, are also available for the majority of shared memory multiprocessors. Portability of applications between different systems is a prime concern of software developers. This, combined with the lack of standard shared memory parallel API, means that, for the most part, application developers use the message passing model, even when the target computer systems are all shared memory systems. A basic aim of OpenMP is to offer a portable standard parallel API that is designed specifically for programming shared memory multiprocessors [46]. MPI is highly portable and is specifically optimized to use with most hardware [1]. In addition, it is suitable to use on a wider range of problems than OpenMP. MPI can also handle larger amounts of data, as it can be distributed among multiple processors. This would be impossible with the memory space available on SMP [42, 95].

MPI may well be the best choice if the application is to be portable on clusters and SMP machines, but if more than eight or sixteen cores are not going to be used, it is advisable to choose OpenMP.

As stated in Chapter 4, the implementation for the computation parts of MPI and OpenMP were kept as close as possible for the process of reading words, parsing letters, and concatenating and computing page numbers.

For the concordance implementation, the SPMD programming model is achieved in OpenMP and MPI using the worker ID. In the case of OpenMP, the Fork/Join model is used, where the master forks to the specified numbers of workers. For MPI, the message passing paradigm is used through the Master/Worker pattern, where the master is sending works and combining results.

During the parallelization attempt, several challenges were encountered, including decomposition of data, choosing the best granularity, minimizing the different kinds of overhead, debugging, and profiling for optimized performance. The aforementioned
challenges were addressed differently, using either a mid-level (MPI) or a high-level (OpenMP) approach.

Solutions to common problems were implemented in a similar way, such as finding sequences across worker boundaries, dividing the file, and specifying the offset for each worker. OpenMP is considered to be a high-level parallel programming approach that is generally easy to use because of its incremental approach and ease of parallelism. However, it offered no advantages for this project due to the nature of the parallel concordance algorithm, which required the use of the SPMD model, thereby increasing programming effort. Consequently, all coordination aspects such as communication, thread management, and synchronization were managed explicitly among threads.

MPI on the other hand offers medium pain parallelism, since the programmer has to manage the parallelism explicitly. For the concordance program, considerable programming effort was required to implement the packing and unpacking of the hash table, since the hash table consists of different elements with different data types. These are keys, values and linked lists representing the page numbers. Moreover, dividing the hash table into smaller chunks, achieving best possible granularity and reducing the communication overheads, were all difficult tasks. Table(5.8) shows the code size for each version of the concordance benchmark.

<table>
<thead>
<tr>
<th>Sequential</th>
<th>OpenMP</th>
<th>MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Haskell</td>
<td>Naive</td>
</tr>
<tr>
<td>256</td>
<td>74</td>
<td>447</td>
</tr>
</tbody>
</table>

The considerable programming effort required for OpenMP is less than for MPI as indicated by the Naive code size for both in Table(5.8) as a direct benefit from the shared memory. However, the tuned OpenMP code is about 19 times longer than the MPI counterpart. Approximately 90% of the tuned OpenMP code is for reducing the synchronization overhead by managing access to the large number of hash tables.

5.4.2 Performance Comparison

Obviously, applications are the best test of the performance of hardware, but interesting information could be obtained by comparing the same program written in MPI and OpenMP [42].

Table(5.9) summarizes the runtime (R), speedup (S), and efficiency measured for C+MPI and OpenMP on different number of cores for experiment 2.
Table 5.9 Final runtimes, speedups, and efficiency for MPI and OpenMP for E2

<table>
<thead>
<tr>
<th>Number of Cores</th>
<th>OpenMP Runtimes</th>
<th>OpenMP Speedup</th>
<th>OpenMP Efficiency</th>
<th>MPI Runtimes</th>
<th>MPI Speedup</th>
<th>MPI Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>54.16</td>
<td>0.8</td>
<td>1.000</td>
<td>67.3</td>
<td>0.6</td>
<td>0.34</td>
</tr>
<tr>
<td>2</td>
<td>28.8</td>
<td>1.6</td>
<td>0.940</td>
<td>67.3</td>
<td>0.6</td>
<td>0.34</td>
</tr>
<tr>
<td>3</td>
<td>20.1</td>
<td>2.3</td>
<td>0.898</td>
<td>33.52</td>
<td>1.3</td>
<td>0.46</td>
</tr>
<tr>
<td>4</td>
<td>15.7</td>
<td>2.9</td>
<td>0.862</td>
<td>23.0</td>
<td>2.0</td>
<td>0.50</td>
</tr>
<tr>
<td>5</td>
<td>12.9</td>
<td>3.6</td>
<td>0.839</td>
<td>18.8</td>
<td>2.4</td>
<td>0.49</td>
</tr>
<tr>
<td>6</td>
<td>11.1</td>
<td>4.1</td>
<td>0.813</td>
<td>18.6</td>
<td>2.5</td>
<td>0.41</td>
</tr>
<tr>
<td>7</td>
<td>9.8</td>
<td>4.7</td>
<td>0.789</td>
<td>18.3</td>
<td>2.5</td>
<td>0.36</td>
</tr>
<tr>
<td><strong>8</strong></td>
<td><strong>8.7</strong></td>
<td><strong>5.3</strong></td>
<td><strong>0.778</strong></td>
<td><strong>18.0</strong></td>
<td><strong>2.6</strong></td>
<td><strong>0.32</strong></td>
</tr>
<tr>
<td>Sequential Runtime</td>
<td>46.5</td>
<td>-</td>
<td>-</td>
<td>46.5</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

The master/worker model is used in MPI so, running the MPI program with single core, was not feasible. This is because the MPI program was designed to have one master, which is responsible for sending work and combining results without being involved in the computational work, Therefore, the measurements for MPI started from double cores.

As shown in Table(5.9), sequential programs are slightly faster than their parallel counterparts on a single processor (OpenMP) or on double processor (MPI) due to parallel overhead, added by MPI library and OpenMP directives.

Figure (5.10) depicts the measured runtime with the number of cores for the final MPI and OpenMP implementations for Experiment 2 compared with the sequential version. Figures(5.11) depict the speedup for the final OpenMP and MPI implementations of Experiment 2.

The more available processors there are, the more the runtime of the parallel versions decrease. The rate of decrease in the runtime directly affects the efficiency, which is an indicator of scalability, 1 being the highest level of efficiency.

The results of the parallel concordance benchmark experiment demonstrate that OpenMP provide outstanding performance with the speedup of 5.3 on 8 cores. However, the price of this performance is the considerable programming effort needed in order to reduce the synchronization overhead. This was achieved by dividing the hash table into more than 600 hash tables, and using critical constructs with different
names for each one.

MPI still provided acceptable performance under the same conditions with a speedup of 2.6 on 8 cores. Considerable efforts were made to improve the MPI version. A great improvement was achieved when the MPI implementation was changed from MPICH to MPICH2, with 30% reduction in the runtime, as a result of the improved communication on shared memory architectures provided by MPICH2. In MPICH1, the communication was achieved through double memory copying in order to pass message from sender to receiver, which is obviously involved extra overhead, whereas MPICH2 efficiently supported communication on multi-core architecture by using Nemesis as a communication subsystem \([30, 31]\).

For all versions, the performance was increased significantly as the computation size increased where the workers were spending more time on computation than on com-
munication (packing and sending in MPI or updating the hash table in OpenMP, as shown in experiment 2 measurements results which were presented in the previous section.

As anticipated, OpenMP showed excellent results, outperforming the MPI version for the parallel concordance benchmark. There was a clear advantage in using shared memory where there was no effect of communication overhead. Moreover, the scalability of the OpenMP version was superior to that of MPI, as indicated by the efficiency values for both. MPI paid an important performance penalty as a result of the communication cost. As the communication to computation ratio increased, the MPI performance decreased significantly, as shown in table (5.9) In addition to communication cost, Additional overhead was involved in order to pack/unpack the hash table for communication.

5.4.3 Summary

OpenMP showed excellent results, outperforming the MPI version for the parallel concordance benchmark. It is about twice as fast as the MPI version, and there was a clear advantage in using shared memory as it avoids the problem of communication overhead.

OpenMP delivers better performance with both experiments:
- For experiment 1, the file size is 18mb, a speedup of 3.3 on 8 cores was achieved, while MPI achieved only 1.3.
- The overall speedup increased for both versions as the computation increased: in experiment 2 (file size = 1mb and N = 50), with 5.3 of the OpenMP and 2.6 of the MPI on 8 cores.

OpenMP shows better scalability than MPI, as indicated by the efficiency values for both.

Although the code resulting from performance tuning of the OpenMP version is about 19 times longer than the MPI counterpart, the effort required to tune OpenMP version is less than that required for MPI, as a direct benefit of the shared memory.

The main disadvantage of the OpenMP version is: the length of the resulting code, where over than 90% of it is for reducing the synchronization overhead by dividing the hash tables to over than 600 hash tables. The code produced is hard to debug and prone to error.

MPI delivers shorter code than OpenMP. The disadvantages are as follows:
- The high communication cost required to send the hash table increases the overhead, leading to poorer performance.
- Considerable effort was required to parallelize the concordance using MPI for packing and unpacking the hash table and dividing the messages into smaller chunks, finding the best granularity and computing the buffer’s and chunk sizes.
Chapter 6

Conclusion

This chapter presents the conclusions reached for this project. Section (6.1) is a summary of the work carried out and presented in this dissertation. Section (6.2) outlines the limitations of the parallel benchmark developed in the project. Section (6.3) presents plans for continuing the research in future work.

6.1 Summary

In this dissertation, the work of constructing a new parallel benchmark, a parallel concordance program, and making comparative evaluations on multi-core architecture is presented.

The dissertation begins with a literature survey of concordance and its history and importance to text analysis, as well as parallelism, in terms of parallel architecture types, parallel architecture trends, parallel programming approaches, models and technologies:(Chapter 2).

Following that, the design and implementation of the two sequential versions in C and Haskell are presented, as well as solutions to some interesting problems. When a comparative evaluation of the performance of the sequential versions was carried out, it was found that the C version was approximately three times faster than the Haskell counterpart as a result of the high abstraction overhead of the latter. For the same reason, Haskell code was more than three times shorter:(Chapter 3).

Next, the parallel concordance in MPI and openMP that was designed for this project is presented, with a discussion of some interesting problems and the solutions found:(Chapter 4).

A description is given of how the naive versions were profiled and benchmarked on multi-core architecture using different profiling tools for each version: ompP for
OpenMP, and mpiP for MPI, and some different ways of reducing different kinds of overheads are discussed. Tuned parallel OpenMP and MPI in C versions of the concordance were also developed. In addition, a comparison of parallel programming languages was made in terms of programming effort, length of code, speedup, and efficiency. A comparison of the performance of the OpenMP and MPI concordance programs on 8 core architectures show that the openMP version achieved a speedup of 5.3 on the 8 cores for a file size of 1MB and sequence length up to 50 words while the MPI version achieved a lesser speedup of 2.6 as a result of higher communication costs. This is because in order to send the hash table in MPI, it must be packed into a buffer and copied from the sender’s buffer to the receiver’s buffer. In OpenMP, however, the memory is shared and no copying of data is required. On the other hand, in order to reduce the synchronization overheads by 92%, the tuned openMP version needed over 600 hash tables.

Several alternatives have been investigated to improve the MPI performance, including: compressing data for communication, replacing blocking to non-blocking communication, and grouping data for communication. The main obstacles to great speedup in MPI were due to the fact that the previous MPI implementation (MPICH1), does not take full advantage of the shared memory architecture, because it copies the data twice, creating unnecessary overheads. With the newer implementation of MPI (MPICH2), the performance was improved significantly by improving the message passing over multi-core architecture.

These results show that openMP delivered better performance with all the experiments, and that despite the MPI program being shorter than the OpenMP counterpart, the OpenMP program is more readable and easier to understand. MPI paid an important performance penalty as a result of the communication cost. More effort was required when developing the MPI version, for packing and unpacking the hash table and dividing the messages into smaller chunks, and finding the best granularity, whereas OpenMP version has the direct advantage of shared memory, and less effort was required to reduce the synchronization overhead: (Chapter 5).

6.2 Limitations

The parallel concordance benchmark developed in this project has a number of limitations:

- The Glib Hash tables [21] are well designed and efficient. However, their capacity is limited, and they can only be used for average memory usage [6].

- The parallel versions of openMP and MPI were tested on a single multi-core architecture with only 8 cores. It would be useful if the programs could be tested on machines with more cores, such as many-core machines to take full advantage of these cores in order to achieve higher performance.
• The limited available versions of GHC compiler does not include the version 6.12.1 which has been proved to fix the hash table performance, and due to the memory limitation, it has not been installed. The sequential Haskell version, therefore, was tested on another multicore architecture, double core machine as stated in Chapter 3.

• The only implementations of MPI that were available were MPICH1 and MPICH2-1.2.1p1. It would be useful to test the MPI version using MPICH2-1.3b1 [10], as it is the latest release of the MPI implementation.

6.3 Future Work

For future work, following the interesting results obtained from this research, it would be useful to:

• Proceed to the next step by parallelizing the sequential Haskell version of the concordance program using GpH (or some other parallel Haskell e.g. Eden, Data parallel Haskell) as a high level parallel technology and carrying out a comparative evaluation with the MPI and OpenMP versions in terms of programming effort, performance and scalability.

• Further research and detailed analysis could be carried out in order to improve the performance of MPI version, such as using single data copying protocol for communication on multi-core architecture proposed by Miao et al [74], or reducing communication time through message pre-fetching [62], as well as running the MPI version on the latest MPI implementation which is MPICH2-1.3b1 [10].

• Further improvement could be made to the openMP version in order to reduce the code size such as using a multidimensional array of hash tables instead of having large number of hash tables.

• It would also be interesting to proceed to larger, different multi-core architecture with more cores, or to the newest many-core [36,53,59], in order to take advantage of these cores to increase the performance.

• In addition, it would be interesting to benchmark the MPI version and GpH-GUM on Bewoulf, and carry out a comparative evaluation between MPI and GpH.

• Since the parallelization work has already been done on the most intensive computation work, which is constructing the concordance, further parallelization could be carried out in future for other computations such as sorting the hash table.
Bibliography


[77] P. Pacheco, Parallel Programming with MPI, Morgan Kaufmann, October 1996.


Appendix A

Project Plan

Here is my project plan which shows clearly the stages involved in order to develop the two concordance benchmarks.

<table>
<thead>
<tr>
<th>Task</th>
<th>March</th>
<th>April</th>
<th>May</th>
<th>June</th>
<th>July</th>
<th>August</th>
</tr>
</thead>
<tbody>
<tr>
<td>Literature Review</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B1 sequential code</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B1 C with MPI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B1 openMP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B1 GPH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B1 Performance analysis</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B2 sequential</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B2 C with MPI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B2 openMP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B2 GPH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B2 Performance Analysis</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Documentation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Submission</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure A.1 Project Plan

You can put extra content in an appendix.