ON THE PERFORMANCE CHARACTERIZATION AND EVALUATION OF RNA STRUCTURE PREDICTION ALGORITHMS FOR HIGH PERFORMANCE SYSTEMS

S. P. T. KRISHNAN

(*M.Sc., National University of Singapore*)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2011

Acknowledgments

It is a pleasure to thank the many people who made this thesis possible.

First, it is difficult to overstate my gratitude to my Ph.D. supervisor, Assoc. Prof. Bharadwaj Veeravalli. His enthusiasm, inspiration, and his great efforts to explain things clearly gave me the confidence to explore my research interests; his guidance helped me to avoid getting lost in my exploration. Throughout my thesis-writing period, he provided encouragement, sound advice, good teaching, good company, and lots of good ideas. I would have been lost without him and this thesis would not have existed in the first place.

I would like to express my sincere gratitude to Prof. Vladimir Bajic (KAUST) for introducing me to the world of cell biology.

I would also like to deeply thank Assoc. Prof. S. K. Panda for providing substantial support and inspiration over the years. He has also offered many constructive advices. I am also grateful to Prof. Lawrence Wong for his support and guidance.

I would like to express my gratitude to my employer Institute for Infocomm Research (I^2R) for supporting me during this part-time study.

I wish to thank Mr. Jean-Luc Lebrun who helped to horn my technical writing skills.

I would also like to acknowledge the efforts of the following former undergraduate students who helped by conducting additional experiments and cross-validating the results - Derrick, Sze Liang, Zhi Ping, Yong Ning, Mushfique, Guangyuan, Hashir, Keith Loo, Praveen and Soundarya.

The thesis marks the end of a long and eventful journey for which there are many people that I would like to acknowledge for their support along the way. Above all I would like to acknowledge the tremendous sacrifices that my parents, Dr. S. K. Padmanabhan and Mrs. S. P. Tarabai, made to ensure that I had an excellent education. For this and their support, love and encouragement I am forever in their debt.

Finally, I would like to thank my wife Kavitha for her endless love, understanding, support, patience, and sacrifices that gave me the bandwidth required to make this journey possible. Without her I would have struggled to find the inspiration and motivation needed to complete this thesis. Special thanks to my daughter Balini Bhadra for letting me write my thesis and understanding that daddy is busy. It is to my parents, wife and daughter, I dedicate this thesis.

Contents

Summary

Scientific problems in domains such as bioinformatics demand high performance computing (HPC) based solutions. Yet, many of the existing algorithms were designed during the era of single-core CPU computing. These algorithms have traditionally benefitted from the performance scaling of the single CPU, typically through higher CPU clock speeds, with no code changes. Currently, the trend among processor manufacturers to get performance scaling is to add additional computing cores rather than make the individual cores more powerful. This requires that the existing algorithms be redesigned in order to run efficiently in this new generation of parallel computers. It also emphasizes the need that parallelization should be considered at the design stage itself, so that new algorithms can scale from single-core computers to many-core computers automatically.

In this thesis, we design and analyze several parallelization methods, and apply them to highly recursive dynamic programming based RNA secondary structure prediction algorithms. We have implemented the parallelized versions of the algorithm on three different high-performance-computing architectures. By conducting

large-scale experiments using different system configurations in these three architectures, we are able to characterize the performance trends on today's parallel computers. The parallelization techniques that we have explored and used are - data parallelization, including wavefront parallelization, code parallelization and hybrid parallelization.

The three high-performance-computing architectures that we have used in our experiments are the Intel x64, IBM Cell Broadband Engine and the Google App Engine (GAE). Each of these systems were chosen because of their respective uniqueness. The Intel architecture is a homogenous ISA (Instruction Set Architecture) multi-core system of Uniform Memory Access (UMA) type, while the Cell is a heterogeneous ISA multi-core system of Non-Uniform Memory Access (NUMA) type. GAE is a task-based multi-system parallel computing platform that is highly scalable for extreme amounts of workloads.

Secondly, we designed a novel parallel-by-design RNA secondary structure prediction algorithm. The algorithm has been designed such that it does not contain any features that will inhibit the parallel execution of the algorithm. The algorithm is designed to scale from single-core to many-cores automatically. We have implemented optimized versions of this algorithm on the three HPC architectures described above.

Using real RNA primary sequences, we conducted large-scale experiments for both of these algorithms on the mentioned three HPC hardware architectures. We modified the system configuration and repeated the experiments for each of these archi-

tectures. This resulted in the generation of large number of data points, comprising of program runtimes and other performance metrics. We subsequently analyzed this dataset and computed the performance trends such as Speedup, Incremental Speedup and Performance gain. The large-scale study has helped in identifying the best possible parallelization technique that can be used to parallelize existing Dynamic Programming based highly recursive algorithms. It has also helped in identifying the performance bottlenecks, system limits and programming challenges of the various high performance computing systems.

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Nucleic Acids

Molecular biology is the branch of biology that deals with the molecular basis of biological activity. Molecular biology chiefly concerns itself with understanding the various systems of a cell and the interactions between them.

Nucleic acids are the most important biological macromolecules and include DNA (deoxyribonucleic acid), RNA (ribonucleic acid) and Proteins. All living cells and organelles contain both DNA and RNA, while viruses contain either DNA or RNA, but not usually both. Nucleic acids consist of a chain of linked units called nucleotides, each of which contains a sugar (ribose or deoxyribose), a phosphate group, and a nucleobase. There are four types of nucleobases in DNA - Adenine (A), Cytosine (C), Guanine (G), and Thymine (T). RNA contains the base Uracil

(U) in place of Thymine. As nucleic acids are non-branched polymers they can be written as a sequence of letters specifying the sequence of nucleobases.

Naturally occurring DNA molecules are double-stranded. James D. Watson and Francis Crick determined the structure of DNA [98] using the x-ray crystallography that indicated DNA had a helical structure (i.e., shaped like a right-handed corkscrew). The double-helix model has two strands of DNA with the nucleotides pointing inward, each matching a complementary nucleotide on the other strand. Nucleotides 'A' and 'T' pair together, and nucleotides 'C' and 'G' pair together. These base pairs are typically called as Watson-Crick base pairs. The base pairing between Guanine(G) and Cytosine(C) forms three hydrogen bonds, whereas the base pairing between Adenine(A) and Thymine(T) forms two hydrogen bonds. Thus, in a two-stranded form, each strand effectively contains all necessary information, redundant with its partner strand.

RNA molecules are single-stranded and do not appear as a double-helix structure. Instead, they adopt highly complex three-dimensional structures that are based on short stretches of intra-molecular base-paired sequences [31] that include both Watson-Crick and non-canonical base pairs. An example of non-canonical base pair is the bond between Guanine(G) and Uracil(U).

Nucleic acids have directionality due to the differences in the chemical composition of the bases and are known as the 3' and 5' ends of the molecule. The directionality is vitally important to many cellular processes, such as gene expression, and the primary structure of a DNA or RNA molecule is reported from the

5' end to the 3' end. In molecular biology and genetics, the term 'sense' is used to compare the polarity of nucleic acid molecules, such as DNA or RNA, to other nucleic acid molecules. A single strand of DNA is called the sense strand if an RNA version of the same sequence is translated or translatable into protein. Its complementary strand is called antisense strand. The mRNA sequence is similar to the DNA strand, however the transcription happens on the antisense strand, by complementing the nucleotides. The terms sense and antisense also applies RNA viral genomes, to refer to whether they are directly translatable (like mRNA) into protein or if they need a RNA polymerase to assist in the translation. The cell machinery directly translates the sense viral RNA into viral proteins. For example, the common influenza virus belongs to the class of antisense RNA.

1.2 Gene Expression

The central dogma of molecular biology, first articulated by Francis Crick in 1958, states that information flow is unidirectional from DNA to Protein and never transfers from protein back into the sequence of DNA. The regions of a DNA that are responsible for the start of this information transfer are called as Genes.

Genes are universal to all living organisms. Genes correspond to local regions within DNA. There are two major type of genes, protein-coding and RNA-coding genes [30]. The process of producing a protein from DNA comprises of two major sequential processes - transcription and translation. Transcription is the process

in which a single-stranded mRNA (Messenger RNA) is created from the coding strand of the DNA. Translation that follows transcription is the process in which a protein is assembled using amino acids with mRNA as the template. RNA-coding genes [30] must still go through the first step, but are not translated into protein.

The genetic code is the set of rules by which a gene is translated into a functional protein. Each group of three nucleotides in the sequence, called a codon, corresponds either to one of the twenty possible amino acids in a protein or an instruction to end the amino acid sequence. The genetic code is nearly universal among all known living organisms.

The order of amino acids in a protein corresponds to the order of nucleotides in the gene. The amino acids in a protein determine how it folds into a three-dimensional shape; this structure is, in turn, responsible for the protein's function. Proteins carry out almost all the functions needed for cells to live. A change to the DNA in a gene can change a protein's amino acids, changing its shape and function; this can have a dramatic effect in the cell and on the organism as a whole.

1.3 Molecular Structures

In this context, molecular structures refer to the structure of nucleic acids such as DNA and RNA. It is usually divided into four different levels. The primary structure is the raw sequence of the nucleotides (represented by their nucleobases) in a nucleotide sequence. Secondary structure, as shown in Figures {2.1, 2.2, 2.3,

2.4}, is a two-dimensional structure formed due to the interactions between bases in the nucleotides. Tertiary structure is the three dimensional layout of the secondary structure taking into consideration geometrical and steric constraints. Quaternary structure is the higher-level organization of nucleic acid like DNA in chromatin or interactions between separate RNA units in the ribosome or spliceosome.

1.4 Molecular Structure Determination

In this method, biochemical techniques are used to determine the structure of nucleic acids. This analysis can be used to determine the patterns that can then infer the molecular structure and function. Molecular structure can be probed using many different methods that include chemical probing, hydroxyl radical probing, Selective 2'-Hydroxyl Acylation Analyzed by Primer Extension (SHAPE), Nucleotide Analog Interference Mapping (NAIM), and in-line probing. As can be seen, these methods are both time-consuming and resource-intensive and requires high-level of skill set from an experienced individual.

1.5 Molecular Structure Prediction

In this method, a computational algorithm is used to determine the secondary and tertiary structures from the primary sequence of a nucleic acid such as DNA or RNA. Secondary structure can be predicted from a single [66] or from several

nucleic acid sequences [89]. Tertiary structure can be predicted from the sequence, or by comparative modeling (when the structure of a homologous sequence is known).

There are several important reasons why molecular structure prediction is increasingly used when compared to molecular structure determination. The following lists some of these key reasons.

Expensive Molecular Structure Determination in a biological lab is an expensive process, in terms of both time and financial costs. Therefore, it is important to determine which sequences are worthwhile to be processed in a biological lab as the cell machinery contains a large amount of nucleotides material with unknown functionality.

Large-scale Sequencing In recent years, nucleotide sequences of lot of organisms have been sequenced. It is simply impossible to process all of them. Therefore, the biological community is looking towards the computing community to help quicken the process.

Homologous Sequences It is a well-known fact that animals and plants have similar genetic material. Hence, there is a large likelihood that their nucleic acids are also similar. Therefore, it would make sense to compare the different nucleic sequences and draw inferences on their structure and functions. This can be used to study further in a biological lab.

Alternate Structures It is also known that the same primary sequence folds

into different secondary and tertiary structures under various circumstances. It would be easier to process this in a virtual software-based environment instead of a biological lab.

Visualization Visualizing the three-dimensional structures is very important and is a task that the computers can do easily and repeatedly when compared to a technician in a biological lab.

1.6 RNA Secondary Structure Prediction

There are minor differences in the approaches to RNA and DNA structure prediction. In vivo, DNA structures are more likely to be duplexes with full complementarity between two strands, while RNA structures being single-stranded and therefore unstable are more likely to fold into complex secondary and tertiary structures. At the molecular level, the extra oxygen in RNA increases the propensity for hydrogen bonding in the nucleic acid backbone. The problem of predicting nucleic acid secondary structure is therefore dependent mainly on base pairing and base stacking interactions. The energy parameters are also different for the two nucleic acids - DNA and RNA.

A common problem dealing with RNA is to determine the three-dimensional structure of the molecule given just the nucleic acid sequence. Moreover, in the case of RNA much of the final structure is determined by the secondary structure or intra-molecular base-pairing interactions of the molecule. This is shown by the

high conservation of base-pairings across diverse species. Secondary structure of small RNA molecules is largely determined by strong, local interactions such as hydrogen bonds and base stacking. To predict the folding free energy of a given secondary structure, an empirical nearest-neighbor model is usually used. In the nearest neighbor model the free energy change for each motif depends on the sequence of the motif and of its closest base pairs. The model and parameters of minimal energy for different nucleotide pairs and loop regions were derived from empirical calorimetric experiments. Summing the free energy for such interactions normally provides an approximation for the stability of a given structure. There are several types of secondary structural motifs and the most complex amongst them is pseudoknots. Many secondary structure prediction methods rely on variations of dynamic programming and therefore are unable to efficiently identify pseudoknots.

1.7 Motivations for our Work

The following are the major motivations for us to undertake this research work:

- RNA structure prediction is common to both Protein-coding and RNA-coding (or non-coding) genes. Therefore, our work will have a wide impact as it is applicable to both the genetic code pipelines.
- RNA tertiary structures are closely related to the secondary structures and are highly dependent on the accurate and quick prediction of the secondary

structures. Therefore, our work on predicting secondary structure can be useful in determining the three-dimensional structure as well.

- RNA secondary structure prediction using computational methods are valued because determination of secondary structures, particularly for long-chain RNA molecules, is difficult by experimental means.

- Many of the existing RNA secondary structure prediction algorithms are based on dynamic programming; refer to Section 2.4.1. Consequently, they are not able to predict pseudoknots completely or do not predict major and important sub-classes within them. Therefore, there is a need for a new algorithm that need not demarcate secondary structures prediction along the boundaries of pseudoknot and Non-pseudoknot.

- The computing paradigm is undergoing a radical change from single-core computers with higher CPU clock speeds to multi-core parallel computers with lower CPU clock speeds. This means that existing iterative algorithms such as those based on dynamic programming will be inefficient (as they cannot use additional computing cores) and therefore slow in producing the results. At the same time, the molecular sequencing efforts is on the rise to sequence all or most of the organisms in earth. Therefore, it is important that existing algorithms be made scalable and fast so that they can be deployed on a large-scale.

1.8 Contributions & Scope of this Thesis

This thesis is primarily concerned with the performance evaluation and characterization of parallelized algorithms on high performance computing systems. The domain we have chosen is bio-informatics and in particular RNA secondary structure prediction. Our primary objective is to parallelize an existing sequential algorithm on HPC architectures and study the performance gains and trends. We chose the PKNOTS [76] algorithm for two reasons - it is one of the leading (and highly cited) RNA secondary structure prediction algorithm and also because it was available freely in source code form. We have developed optimized versions of PKNOTS on three HPC architectures. We limit our validation efforts by comparing the output of our parallelized versions to the original unmodified sequential version only. Specifically, we do not validate the predicted structures of PKNOTS. Subsequently, using this experience we have designed a new RNA secondary structure prediction algorithm MARSs. Unlike PKNOTS, MARSs is a non-iterative algorithm and is expected to run efficiently on both single-core and multi-core architectures. As MARSs is a new algorithm we have compared the output of MARSs to that of known structures for corresponding primary sequences and show that MARSs is capable of predicting high-quality secondary structures. We collected a large dataset of actual RNA primary sequences and used it in our large-scale experiments. All the sequences have known secondary structures and have both pseudoknots and non-pseudoknots. We conduct large-scale experiments for both PKNOTS and MARSs under multiple system configurations and observe

their respective performance characteristics.

1.9 Organization of this Thesis

Rest of this thesis is organized into the following chapters:

Chapter 2 This chapter covers the background material. More specifically, we conduct detailed literature surveys into existing & leading RNA secondary structure prediction algorithms. These algorithms are based on diverse methodologies such as dynamic programing, comparative search and heuristics. Following this, we discuss about generic parallelized algorithms, parallelized RNA structure prediction algorithms and the parallel computing landscape.

Chapter 3 In this chapter we describe the PKNOTS RNA secondary structure prediction algorithm. We then analyze the algorithm, identify performance hotspots and parallelize the software implementation. We evaluate different parallelization methods and share & discuss the strengths & weaknesses of them. We also provide early results using a small-scale dataset as well.

Chapter 4 This chapter introduces the new algorithm that we propose as part of this thesis. We describe the algorithm step-by-step and in detail for the reader to understand. We then describe a set of quality measures and show a predicted example using our algorithm.

Chapter 5 This chapter contains details on the experiments performed and the results generated. We parallelize PKNOTS on 3 different parallel hardware architectures and discuss the customizations required & optimizations performed. We also implement our MARSs algorithm on the same parallel architectures and discuss the results from the two large-scale experiments.

Chapter 6 In this chapter we conclude this thesis by summarizing our contributions and also share the plans for the short-term enhancements and suggestions for the long-term improvements.

Chapter 2

Background

2.1 Introduction

There are two main objectives for this chapter. First, we describe in detail the RNA secondary structure prediction process from a computational perspective. In this section, we show the need for High-Performance Computing (HPC) approaches for RNA secondary structure prediction. Second, we discuss several RNA secondary structure prediction algorithms and highlight their strengths & weaknesses from the perspective of predicting the different types of RNA secondary structure motifs, time & space complexities and their suitability of being ported to a HPC architecture.

2.2 RNA Secondary Structure Prediction

As briefly mentioned in Chapter 1, the central dogma of molecular biology highlights the fact that the protein-production transaction is RNA-mediated. In addition, RNA is involved in both coding (i.e., making protein as end product) and non-coding (i.e., making RNA as the end product) gene expression pipelines. RNA exists in three structural forms - primary, secondary and tertiary - and progressively evolves from the primary to tertiary. In the case of RNA, the tertiary structure closely resembles the secondary structure and therefore predicting correct secondary structures is a key factor in determining the structure and function for both coding and non-coding RNAs.

A secondary structure is formed when nucleobases (or simply bases) in nucleotides form base pairs with complementary bases in other nucleotides. In the case of DNA, the base pairs occur between bases in nucleotides from two different strands. On the other hand, RNA being single-stranded the base pairs occur between nucleotides of the same strand. In case of DNA the purpose of forming base pairs is primarily to replicate the genetic material for preservation and gene expression. In case of mRNA, the purpose is to create a template that is then used in the synthesis of amino acids, the building blocks of proteins.

A RNA secondary structure can be seen as comprising of several structural motifs (or patterns). These structural motifs were discovered through biological (or wet-lab) experiments. A RNA secondary structure is formed when the primary

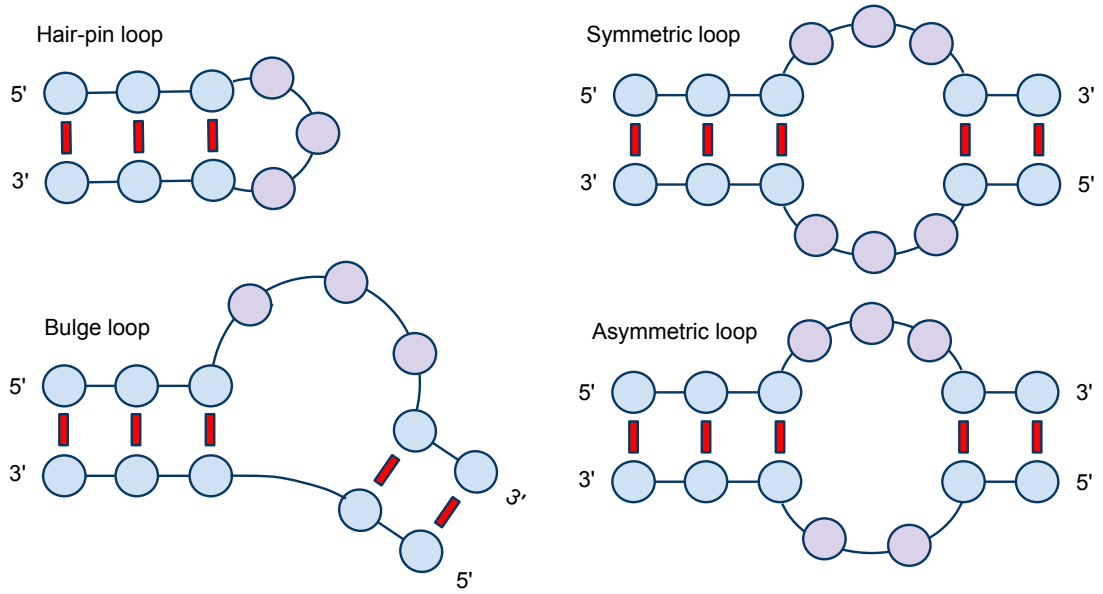


Figure 2.1: RNA Secondary Structure Motifs - Loops

structure (or sequence) folds upon itself resulting in base pairs between compatible & selected free nucleotides. Secondary structural motifs can be classified into two broad categories depending on the number of times a sequence folds upon itself.

The first category "stems & loops" comprises of a set of secondary structural motifs that are formed when the primary structure/sequence folds upon itself once. The different secondary structure motifs are shown in Figures 2.1 & 2.2 and explained below.

Loops are a major type of secondary structural motifs and are closely related to stems. They can be classified into internal loops, bulges and hairpin loops. The different types of loops are shown in Figure 2.1.

Internal loops are of two types - symmetric and asymmetric internal loops. Internal loops are formed when nucleotides interlocked by a steam on either

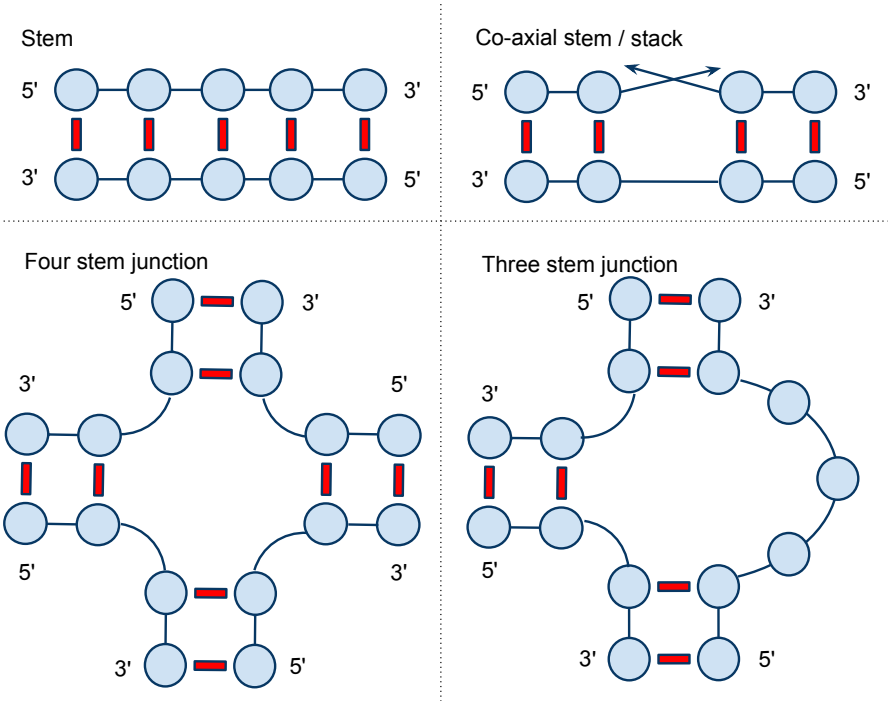


Figure 2.2: RNA Secondary Structural Motifs - Stems & Junctions

side do not form a base pair. When the number of non-pairing nucleotides is same on both the strands the resultant internal loop is called as symmetric internal loop. An asymmetrical internal loop is formed when the number of non-pairing nucleotides on one side of a secondary structure is different from the number of nucleotides on the other side of the secondary structure.

Bulge is a special type of internal loop and is formed when nucleotides are unpaired on only one side of base pairing stem in a secondary structure.

Hair-pin loop is formed when a set of free nucleotides is locked by a single base pair unlike internal loops & bulges that are bounded by two different base pairs. In addition, a hairpin loop is usually formed near a folding point, unlike internal loops & bulges that are always surrounded by base-pairing

stems.

Stem is a type of secondary structural motif that is formed when base pairs are formed across a sequence of nucleotides that are facing each other as a result of the structural fold. There are no free nucleotides (or simply gaps) in-between the base pairs. There are two key attributes for a stem - the length of the stem and the quality of the base pairs. The quality of the base pairs is determined if they are canonical (such as Watson-Creek) or non-canonical (such as Wobble). Stem is shown in Figure 2.2.

Junctions are intersections that are formed when several branches, each comprising of a set of motifs meet at a common point. There can be more than one junction in a secondary structure and each junction can be of different sub-type. There are currently three well-studied junctions - three-stem & four-stem junctions and co-axial stem/stack. A co-axial stem/stack is a tertiary structure and is derived from a four-stem junction. The three junction types are shown in Figure 2.2.

Pseudoknots are the second category of secondary structural motifs that are formed when a primary structure folds upon itself twice in opposite directions. A pseudoknot comprises of at least three secondary structural motifs - loop, stem and a free dangling end. The stem (or at least a single base pair) locks the loop and the free dangling end folds back. Base pairs are formed with the free nucleotides in the hairpin loop or with the free nucleotides interspersed between stems.

Pseudoknots are classified as simple and generic pseudoknots. A simple pseudoknot is formed when the free-dangling end forms base pairs with free-nucleotides in the loop region only. Therefore, a simple pseudoknot usually contains two motifs regions - loops, stems - and could optionally include free-dangling ends. In a generic pseudoknot base pairs are dispersed and interspersed between stem regions as well. Therefore, a generic pseudoknot could contain internal loops (asymmetric, symmetric) and bulges. Figure 2.3 shows both a simple pseudoknot and a generic pseudoknot.

Special Structures In addition to the single-sequence secondary structure folding, it is also possible for base pairs to occur between two independent secondary structures. This is due to the availability of free nucleotides (usually in the loops, bulges) of two secondary structures that are close to each other (in atomic scale). Figure 2.4 shows two examples of such a possibility. We restrict the scope of this thesis to predict single-sequence based secondary structures only.

2.3 RNA Structure Prediction on HPC Systems

Section 2.2 described the process in which a secondary structure is formed from the primary structure. The nucleotides in the RNA primary structure is distinguished by their nucleobases sub-units and abbreviated as A, C, G and U. Assuming a random distribution of nucleotides in the primary sequence, and with the RNA

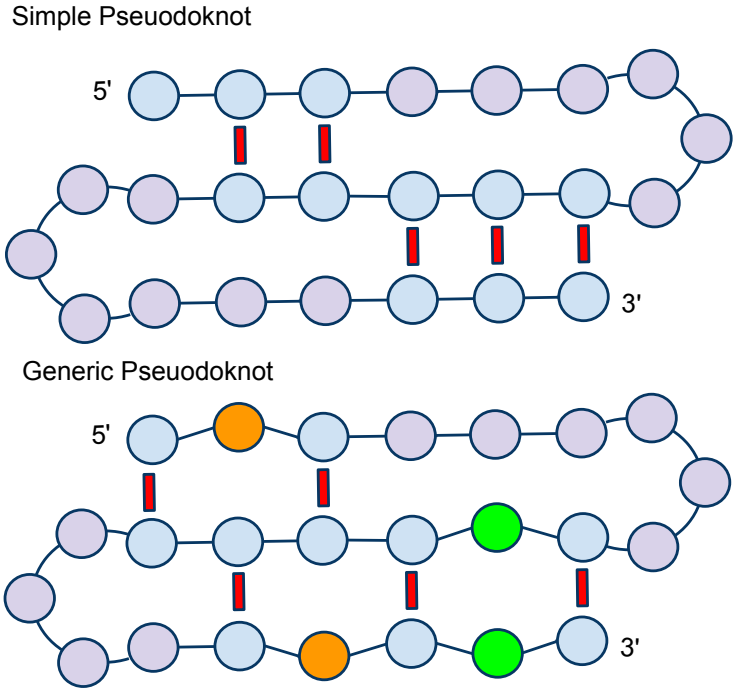


Figure 2.3: RNA Secondary Structural Motifs - Pseudoknots

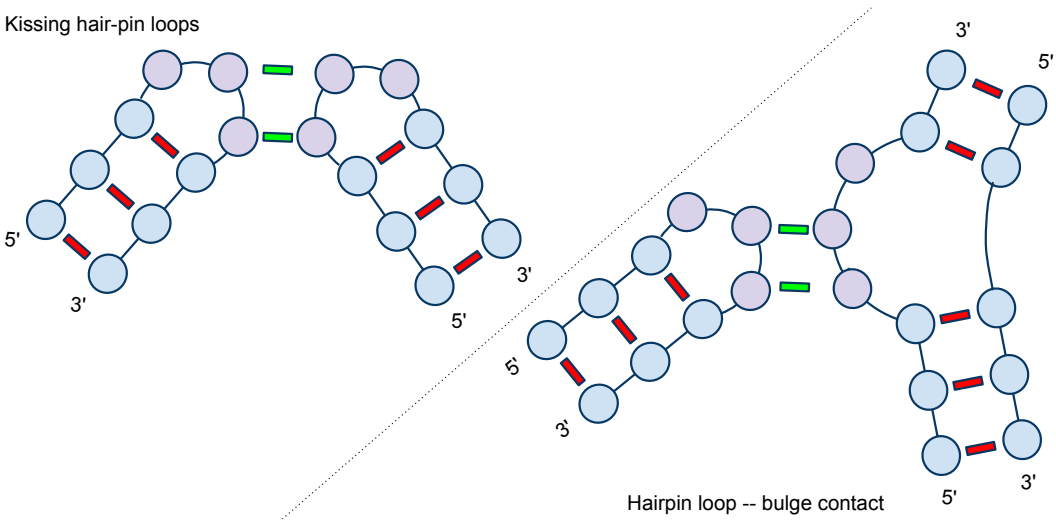


Figure 2.4: RNA Secondary Special Structural Motifs

alphabet size being small, the possibility of base pairs between compatible nucleotides is high, since finding another nucleotide of same type is rather easy. Therefore, it is possible to have more than one RNA secondary structure for a primary structure. This property distinguishes RNA from DNA.

Before the advent of general-purpose computers, RNA secondary structures were exclusively determined using biophysical methods in a laboratory. This method when used exclusively has a couple of shortcomings. First, although the biophysical method is conclusive in determining the secondary structure, it is very expensive from both time and resource perspectives. Second, the method captures or snapshots RNA secondary structure at one point-in-time only. Third, should there be an error in sequencing the primary structure, the process has to be repeated all over again. Fourth, the knowledge gained from previous experiments, like mapping secondary structure motifs to known primary structure sequence, cannot be re-applied. These prompted the biologists to source for alternate methods that can be used ubiquitously & repeatedly with ease as the first-choice and to use biophysical methods selectively afterward.

For nearly three decades, computers have been used as an enabling technology for bioinformatics. Computers are being used to predict molecular structures including RNA secondary structures for more than a decade now. This is primarily due to the explosion in the number of organisms that are being sequenced and the availability of affordable general-purpose computing resources. Importantly, usage of computers helps in step-by-step inspection and visualization of the folding

process, which cannot be easily accomplished with biophysical methods.

The computing power required for molecular structure prediction like RNA secondary structure prediction is much higher compared to other tasks such as data acquisition, organizing, and classification. Therefore, there is a strong need for high performance computing solutions. In this context, it is important to explore, albeit briefly, the evolution of the various HPC architectures and note the strength and weaknesses of each of them.

The computer revolution started with the invention of the microprocessor and aided by the steady improvements in silicon packaging, the CPU has become more and more compact & powerful over the years. As an example, today's smartphones such as Nexus One from Google have more computing power than the desktops of a decade ago. During the years of evolution, performance from a single processor has been achieved primarily by increasing the CPU clock frequency. This method served quite well until recently when its side effects began to out-weigh the benefits that diminished the gains that can be obtained through higher operating frequencies.

The three prominent side effects are - memory wall, frequency wall and ILP (Instruction Level Parallelism) wall. Memory wall refers to the trend where the CPU speed is increasing at a much higher rate compared to the RAM speed. This leads to a situation where the CPU is idling while waiting for the memory sub-system to fetch data for processing or deliver the results. CPU designers have partly mitigated this situation by using caches. Recent CPUs also use larger and multi-level

caches, with 3 levels being the current maximum. This solution however hides the memory latencies as memory bandwidth is the ultimate bottleneck in performance. This also question the need for CPUs with clock frequencies that are much higher than the inter-connecting bus speed.

The second side effect is the frequency wall. In order to understand this situation, let us formulate the power consumed in a chip mathematically and summarize it in Equation 2.1. In this equation, P represents power, C is the capacitance being switched per clock cycle, V is voltage, and F is the processor frequency (cycles per second) [75]. The rising CPU frequencies means more power is consumed and more heat needs to be dissipated from the surface of the chip. In operation, the temperature of a computer's components will rise until the heat lost to the surroundings is equal to the heat produced by the component, and thus the temperature of the component reaches equilibrium. For reliable operation, the equilibrium temperature must be sufficiently low for the structure of the computer's circuits to remain intact and not meltdown. Therefore, there is an upper limit on the amount of power that can be dissipated and this indirectly restricts the CPU frequency scaling.

$$P = CV^2F \tag{2.1}$$

The third side effect is the ILP wall and is closely linked to the frequency wall. ILP is a measure of the number of operations in a computer program that can

be performed simultaneously. The amount of ILP in programs is very application specific. For example, in fields like graphics and scientific computing the amount can be very high while programs in cryptography exhibit much less parallelism. An ILP wall is set to exist when there is not enough parallelism in a single instruction stream to keep a high performance single-core busy.

Despite these issues, transistor densities are still doubling every 18 to 24 months as per Moore's law. With the end of frequency scaling, these new transistors (which are no longer needed to facilitate frequency scaling) are being used to add extra hardware, such as additional cores, to facilitate parallel computing - a technique that is being referred to as parallel scaling. The end of frequency scaling as the dominant cause of processor performance gains has caused an industry-wide shift to parallel computing.

2.4 Literature Survey on RNA Structure Prediction Algorithms

In this section, we highlight some of the more relevant work done in the field of RNA secondary structure prediction. We discuss some of the important algorithms from multiple dimensions such as prediction method, types of secondary structural motifs predicted and performance metrics such as time complexity, space complexity and prediction accuracies. Our objective is to understand the existing algorithms and their suitability for HPC architectures, in particular multi-core

systems.

RNA secondary structures can be derived using two different methods - single sequence based prediction algorithms and multi-sequence based comparative search algorithms. In single sequence algorithms, the input to the algorithm is the primary structure, for which the secondary structure needs to be determined along with applicable thermo-dynamic models and other auxiliary information. Secondary structure is derived using different types of algorithms and the most popular approach for structure prediction is to predict the lowest free energy structure with a dynamic programming algorithm. In comparative search algorithms, one or more primary sequences with known secondary structures are available for reference in addition to the primary structure with unknown secondary structure. Comparing structural motifs between the known sequences and the unknown sequence does the search for secondary structural motifs.

Several generic algorithmic methods have been adopted from other scientific domains and customized to predict RNA secondary structures. There are two major categories - dynamic programming and heuristic-search based algorithms. Dynamic programming is a method of solving complex problems by breaking them down recursively into simpler problems. It is applicable to problems that exhibit the property of overlapping sub-problems and have optimal sub-structures. Dynamic programming based algorithms employ single-sequence based search method. In heuristics-search based algorithms different types of heuristics are used to determine the secondary structural motifs. Some of the well-known methods that adopt

different forms of heuristics are Genetic Algorithms, Quasi-Monte Carlo Search, Stochastic Context-free Grammar, and Hop-field networks.

In addition, algorithms can also be classified based on types of secondary structural motifs that they predict. The motifs can be classified as non-pseudoknots and pseudoknots. Pseudoknots are the most complex of all the RNA secondary structural motifs. RNA pseudoknots are functionally important in several known RNAs [21]. Pseudoknots occurs in a number of functional RNA sequences ([12], [20]). Plausible pseudo-knotted structures have been proposed by Pleij et al., [70], and confirmed by Kolk et al., [50]. Pseudoknots are classified into simple and generic pseudoknots and are shown in Figure 2.3. Simple pseudoknots, as the name implies, are formed by fewer structural motifs while the generic pseudoknots are more complex by nature. Naturally, the algorithms that predict RNA secondary structures can be classified into three groups - those that predict secondary structures without pseudoknots, those that predict secondary structures with simple pseudoknots and those that predict secondary structures with both types of pseudoknots. It is to be noted that the categories described until now do not necessarily mutually-exclude the various algorithms. It is possible to create hybrid algorithms and several algorithms within a single category have different prediction capabilities as well.

Batenburg [12] has compiled a collection of RNA secondary sequences that contain pseudoknots and called it pseudobase. Pseudobase [110] is an online database containing structural, functional and sequence data related to RNA pseudoknots. Each pseudoknot comprises of the relevant sequence and supporting information

such as the reliability of the data, stem location and accession numbers.

The following subsections list the related publications to our research work in chronological order. We have categorized them into dynamic programming, comparative search and heuristics-based methods. Further, we have also listed some of the research work done in parallelizing existing RNA secondary structure prediction methods.

2.4.1 Dynamic Programming based Algorithms

As early as 1971, scientists were debating and proposing stability models for RNA secondary structure. In particular, two pioneering works were by Ignacio et al., ([86], [87]). The proposed stability model calculates the stability of a folded RNA molecule in terms of its free energy by adding independent contributions from base pair stacking and loop destabilizing terms from the secondary structure. This model has proven to be a good approximation of the forces governing RNA secondary structure formation, thus allowing fair predictions of real structures by determining the most stable structures in the model of a given sequence.

The energy rules are essential in these algorithms and therefore the quality of the result depends strongly on the validity of our knowledge about the values of the energetic parameters. One of the reasons for a deviation of the minimum-energy solution from proven secondary structures is our lack of knowledge about thermodynamic parameters used in the calculations [48].

Based on this thermodynamic model, algorithms for computing the most stable structures have been proposed, for example Nussinov & Jacobson [66] and Zuker & Stiegler [104]. Later Zuker [106] also proposed a method to determine all base pairs that can participate in structures with a free energy within a specified range from the optimal.

The Zuker algorithm, implemented in the programs MFOLD [106] and ViennaRNA [42], is an efficient dynamic programming algorithm for identifying the globally minimal energy structure for a sequence, as defined by such a thermodynamic model ([104], [105], [77]). The Zuker algorithm requires $O(N^3)$ time and $O(N^2)$ space for a sequence of length 'N', and so is reasonably efficient and practical even for large RNA sequences. The dynamic programming based Zuker algorithm was subsequently extended to allow experimental constraints, and to sample suboptimal folds [107]. McCaskill's variant [62] of the Zuker algorithm calculates probabilities (confidence estimates) for particular base pairs. One well-known limitation of the Zuker algorithm is that it is incapable of predicting RNA pseudoknots.

The focus of this class of algorithms is to reduce the free energy in the predicted RNA secondary structures. In particular, Zuker et al., [108] report an upgraded version of their program MFOLD that is capable of predicting RNA secondary structures without pseudoknots. Additionally, they exclude the prediction of base triples and also restrict a hairpin loop to contain at least 3 free nucleotides. The algorithm has been updated to use the newer thermodynamic parameters as well.

Rivas et al., [76] describe a dynamic-programming based algorithm that is able to predict a class of pseudoknots, namely simple pseudoknots. This is the first algorithm to predict optimal (minimum energy) pseudo-knotted RNAs using the standard RNA secondary structure thermodynamic model. The algorithm generates the optimal minimal-energy structure for a single RNA sequence, using standard RNA folding thermodynamic parameters ([33], [78]) augmented by a few parameters describing the thermodynamic stability of pseudoknots and using coaxial stacking energies [97] for both pseudoknotted and non-pseudoknotted structures. The authors have tested their algorithm using different classes of RNAs - tRNAs, HIV-1-RT-ligands and viral RNAs. The test set comprised of several small pseudo-knotted and non-pseudo-knotted RNAs. There are several concerns of the proposed algorithm.

1. The worst-case time and space complexities are $O(n^6)$ and $O(n^4)$, where 'n' is the length of the input primary sequence. Due to the high complexities the algorithm can only be used for smaller RNA sequences, typically sequences that are <= 150 nucleotides in length. This is a severe limitation given that longer RNA sequences are being discovered now such as HIV1 genome whose sequence length is 9229 nucleotides.

2. The authors reported a prediction accuracy of 50% median across the datasets that was used with the accuracies being different for different classes of RNAs. This again is a drawback because the algorithm's output cannot be confidentially used and moreover longer RNA sequences have not been

tested with as well.

3. The algorithm uses recursions heavily, making it difficult to take advantage of newer HPC architectures like multi-core processors, where the total computing power of the system is distributed among multiple, typical slower, multi-core CPUs.

Akutsu [4] analyzed Uemura et al., [95] and found that the usage of tree adjoining grammar was not crucial. However, the parsing procedure was crucial and is intrinsically a dynamic programming procedure. Akutsu re-formulated their method as a dynamic programming procedure without tree adjoining grammar. Akutsu also showed the secondary structure prediction for generalized pseudoknots is a NP-hard problem when using free energy thermodynamics.

Jitender et al., [24] proposed a dynamic programming based algorithm that can predict simple pseudoknots as well. The algorithm was reported to have worst-case time and space complexities of $O(n^4)$ and $O(n^3)$. The authors have validated their algorithm by using the simple-pseudoknot subset of the Pseudobase collection. This subset consists of 169 sequences and is the same set as used by Rivas et al., [76]. The authors have reported a prediction accuracy of 95% with 78% of pseudoknots with correct or almost correct structures.

Although the prediction accuracy of Jitender et al., [24] is significantly higher than Rivas et al., [76], two important questions can be raised. First, how does the algorithm perform on a sequence that does not contain any pseudoknot in

its secondary structure? In other words, does the algorithm assume that there will always be a simple pseudoknot in secondary structure? Second, the selected dataset is rather limited, especially the size of the input sequences are smaller. The largest sequence that has been tested is only 114 nucleotides in size. Therefore, the performance is not known for longer sequences and prediction accuracies for sequences with no pseudoknots have been tested as well.

David Mathews et al., [64] revised a dynamic programming algorithm for predicting RNA secondary structures to include folding constraints determined by chemical modification and to include free energy increments for coaxial stacking of helices when they are either adjacent or separated by a single mismatch. Furthermore, free energy parameters are revised to account for recent experimental results for terminal mismatches and hairpin, bulge, internal, and multi-branch loops. The authors report that the percentage of known base pairs in the predicted structure for certain species increased significantly using modification constraints while it remained at the same level for others.

R. Tyagi and DH. Mathews [92] tested and confirmed the hypothesis that RNA secondary structures can be predicted by free energy minimization using nearest-neighbor thermodynamic parameters. In their experiments, the authors observed that their predictions had more than 50% accuracies when compared with crystalized structures.

2.4.2 Comparative-search based algorithms

Comparative-search is a rather reliable approach for RNA secondary structure prediction in which covarying motifs are identified across primary structures that are multiple sequence-aligned, but from different sequences [99]. Covarying residues are indicative of conserved base pairing in secondary structures. Comparative-search based algorithms are generally known to perform better on longer sequences and are likely to be more robust as existing knowledge is reused.

Fariza et al., [90] proposed a secondary structure prediction algorithm p-DCfold based on comparative-search principles. The proposed method is able to predict all types of pseudoknots. This work is based on an extension to the author's earlier work DCfold [89] in which secondary structures with the exception of pseudoknots were searched. In this work, pseudoknots are searched in several steps, and the authors report very satisfactory results without any false positives in helix prediction.

The primary shortcoming of the algorithm includes the ability to search for only non-interleaved helices. Second, being based on comparative-search this algorithm (and other comparative-search based algorithms) cannot be used to predict secondary structures for sequences with no matching primary structural motifs. Other earlier comparative-search based RNA structure prediction methods are [3], [38], [40] and [36]. We do not analyze these algorithms further, as the focus of this thesis is on single sequenced based RNA secondary structure prediction

methods only.

2.4.3 Heuristic-search based Algorithms

Most methods for RNA folding that are capable of folding pseudoknots adopt heuristic search procedures and sacrifice optimality. Examples of these approaches include quasi-Monte Carlo searches [2] and genetic algorithms ([37], [8]). These approaches are inherently unable to guarantee that they have found the "best" structure given the thermodynamic model, and consequently unable to say how far a given prediction is from an experimentally verified structure.

Abrahams et al., [2] contributed one of the earliest software programs to predict secondary structures including pseudoknots. Their algorithm simulates a hypothetical process of folding and uses published, experimentally verified free energy values to get the optimum secondary structure. As a mark of performance, the authors report that their algorithm is able to fold a 700-nucleotide sequence in just over an hour using only a CPU at 8MHz.

Stormo et al. introduced a different approach to pseudoknot prediction based on the Maximum Weighted Matching (MWM) algorithm ([29], [35], [16]). Using the MWM algorithm, an optimal structure is found, even in the presence of complicated pseudo-knotted interactions & base-triples, in $O(N^3)$ time and $O(N^2)$ space. However, MWM currently seems best suited to folding sequences for which a previous multiple alignment exists, so that scores may be assigned to possible base

pairs by comparative analysis. The authors subsequently improved the same algorithm in [88] to improve the accuracy to filter out spurious base pairs and also used new information such as experimental data, statistical and thermodynamic information to calculate the MWM performance.

Batenburg el al., [8] investigated the possibility of using genetic algorithm for the prediction of RNA secondary structures. The authors use a step-wise selection of most-fit structures that is similar to natural evolution. It is also reported that this process allows for easy interchange of various fitness models.

Brown and Wilson [9] proposed a way to model RNA secondary structural motif pseudoknot. The model is based on intersections of SCFGs (Stochastic Context Free Grammars). The authors have used the proposed model to do a database search to find RNA sequences containing one particular type of RNA pseudoknot.

Kim et al., [49] proposed an algorithm using simulated annealing technique. The algorithm uses a multiple-sequence alignment strategy to align multiple RNA sequences to identify conserved RNA secondary structure and in the process identifies secondary structures in new sequences. The algorithm proposes the construction of an intra-sequence dot matrix. A hit probability is then computed based on these matrices and a score function is defined.

Uemura et al., [95] proposed an algorithm based on tree adjoining grammar. The time complexities of their algorithm depends on types of pseudoknots; it is $O(n^4)$ for simple pseudoknots and at least $O(n^5)$ for the other types of pseudoknots. Al-

though the algorithm can always find optimal structures, tree-adjoining grammars are complicated and impractical for longer RNA sequences.

Lyngso et al., [57] proposed a new heuristics-based method to evaluate all possible internal loops of size at most 'k' in an RNA sequence 's', in time $O(k|s|^2)$; this is an improvement from the previously used method that has a time complexity of $O(k^2|s|^2)$. For unlimited loop size this method improves the overall complexity of evaluating RNA secondary structures from $O(|s|^4)$ to $O(|s|^3)$. The authors have used this method to examine the soundness of setting k = 30, a commonly used heuristic.

Lyngso et al [58] has proved that the general problem of predicting RNA secondary structures containing pseudoknots is NP-hard for a large class of reasonable models of pseudoknots. The algorithm has a time and space complexity of $O(n^5)$ and $O(n^3)$. It is able to predict only certain classes of pseudoknots.

Haslinger et al., [44] proposed and implemented a minimum free-energy folding algorithm. The algorithm has runtime complexities $O(mn^3)$ in time and $O(mn^2)$ in space where 'm' is a constant depending on the structural freedom approved to the pseudoknots. The limitation of this algorithm is that it searches only the simplest type of pseudoknot, the H-type pseudoknot.

Ye Ding and Charles E. Lawrence [23] have proposed a statistical algorithm to sample rigorously and exactly the Boltmann ensemble of secondary structures for a given RNA primary sequence. This is important because a RNA molecule,

particularly a long-chain mRNA, may exist as a population of structures each playing important and different functional roles. Thus, a representation of the ensemble of probable structures is of interest.

The algorithm has two steps - aptly named forward and backward. The forward step of the algorithm computes the equilibrium partition functions of RNA secondary structures using thermodynamic parameters. Using conditional probabilities computed with the partition functions in a recursive sampling process, the backward step of the algorithm quickly generates a statistically representative sample of structures. The algorithm has $O(n^3)$ time complexity for the forward step, $O(n^4)$ time complexity in the worst case for the sampling step, and $O(n^4)$ space complexity.

Deschenes et al., [26] improved the prediction of RNA secondary structures using evolutionary algorithms by adding more information on stacking energies to the thermodynamic energies. They have compared the performance of their algorithm against [66]. The authors have done detailed analysis using real world data, but only with three sets of them. They have found that although EA outperforms [66], their algorithm's accuracy is good at sequences of shorter lengths only.

Dowell and Eddy [25] evaluated several lightweight stochastic context-free grammars for RNA secondary structure prediction. In particular, they implemented nine different small SCFGs to understand the tradeoff between model complexity and prediction accuracy on a benchmark set of RNA secondary structures. They conclude that four SCFG designs have prediction accuracy that is near the current

energy minimization algorithms. They further shortlisted one SCFG in PFOLD that is much simpler than others.

Another approach to predict RNA secondary structures is called maximum expected accuracy structure prediction ([54], [28], [47], [61]). Roughly, maximum expected accuracy structures are structures composed of pairs that provide the maximal sum of pairing probabilities. The pairing probabilities can be derived by machine learning methods or by thermodynamic methods using partition functions. Maximum expected accuracy structures have improved accuracy compared with free energy minimization because it has been observed that highly probable base pairs are more likely to be correctly predicted pairs [64].

Stanislav Bellaousov and David H. Mathews [14] proposed a new algorithm to predict RNA secondary structures with pseudoknots of any topology. Their Probknot algorithm assembles maximum expected accuracy structures from computed base-pairing probabilities in $O(N^2)$ time, where 'N' is the length of the sequence. The performance of ProbKnot was measured by comparing predicted structures with known structures for a large database of RNA sequences with fewer than 700 nucleotides. The percentage of known pairs correctly predicted was 69.3%. The resulting sensitivity is therefore higher than Rivas et al., [76] and the authors have also used longer sequences. However, since the algorithm is based on heuristic-search principles, it will not be suitable to predict on RNA sequences with no known analogous sequences. Secondly, the percentage of predicted pairs in the known structure was 61.3%. This means that the algorithm is also predict-

Table 2.1: Summary of Relevant RNA Structure Prediction Algorithms

Algorithm	Method	Time	Space	Pseudoknots
Zuker [106]	DP	$O(N^3)$	$O(N^2)$	No
Rivas [76]	DP	$O(N^6)$	$O(N^4)$	Simple
Jitender [24]	DP	$O(N^4)$	$O(N^3)$	Simple
Stormo [16]	MWM	$O(N^3)$	$O(N^2)$	Yes
Uemura [95]	TAG	$O(N^5)$	Not Specified	Yes
Lyngso [58]	Heuristics	$O(N^5)$	$O(N^3)$	Restricted
MARSs (this thesis)	Multi-model	$O(N^3)$	$O(N^2)$	Yes

ing close to 40% of additional incorrect base pairs, which may not be suitable for predicting on new RNA sequences.

Bon and Orland [15] have proposed a new algorithm to predict RNA secondary structures with pseudoknots. The authors claim that their method will be able to find minimum free energy structures irrespective of pseudoknot topology. The algorithm significantly improves the quality of prediction at the expense of processing longer sequences. The algorithm is based on Maximum-Weighted-Set (WIS) principles and uses graph theory to represent the various base pairs and selects a subset of all base pairs to form the secondary structure with minimum energy.

Table 2.1 summarizes the most relevant RNA secondary structure prediction algorithms to our work. It highlights the prediction method, time & space complexities and the ability to predict pseudoknots. The purpose of this table is to relate our proposed algorithms MARSs to existing algorithms.

2.4.4 Generic Parallel DP Algorithms

In this section, we highlight some of the relevant work done in the field of generic parallel DP algorithms. The objective is to understand some of the existing parallelization methods through which DP based algorithms have been adapted to parallel computers.

Martins et al., [63] demonstrated that it is possible to parallelize a dynamic programming based algorithm using wavefront parallelization techniques. The authors parallelized a sequence comparison algorithm and used the EARTH (Efficient Architecture for Running THreads) execution environment. EARTH is an event-driven architecture where it is possible to define data dependencies for each individual scheduled thread. The system only schedules the thread for execution when all the data-dependencies are satisfied. EARTH is implemented as application libraries & runtime on top of COTS (Components Off The Shelf) hardware & software. In a desktop OS such as Linux, the threads will block when their data dependencies are not satisfied (for example being read). In such cases, EARTH might have a mild performance gain in that a thread is not allotted until all data dependencies are satisfied. The EARTH scheduler though always maintains a list of scheduled threads and executing threads. One disadvantage of EARTH is that it is not aware of the general system workload; this however is not a major issue as in HPC setup it can be assumed that the worker nodes are dedicated.

On an algorithmic level, the authors observed that using the wavefront paralleliza-

tion method, the number of concurrent parallel processors required is uneven and this is a concern for longer sequences. The communication overhead between large numbers of processors should also be considered. The authors have proposed to sub-divide the computation of the single similarity matrix into rectangular blocks. Through this way, the limited number of execution units (or processors) can be reused to compute various parts of the matrix. This method though introduces serialized phases in a parallel computing environment. EARTH is a good candidate for cloud computing where computation nodes may be instantiated on-demand.

Alves et al., [6] worked on a similar biological sequence comparison problem of using wavefront parallelization to distribute work to a distributed memory parallel computer. They used the beowulf cluster architecture based system, comprising of 64 nodes and each node comprising of 256MB RAM and 256MB swap space. The authors have introduced a parameter 'α' that represents the optimum size of the sub-matrices. Subsequently, the authors experimented with variable sized sub-matrices to optimize the computation & communication overhead and documented their results.

The core contribution of this paper is to show that it is possible to split the input data matrices, such that the individual pieces will fit in each of the worker nodes, typically the RAM. The results however may not be directly extensible to secondary structure prediction scenario, due to the data dependencies among the various elements. In this latter case, the CPUs may stall periodically and cache flushes may become frequent during data fetch, thereby limiting the total

performance.

Anvik et al., [5] introduced a method of automatically generating parallelized framework code for wavefront design patterns. They have implemented their proposal as a GUI application in CO_2P_3S (Correct Object-Oriented Pattern-based Parallel Programming System), which generates the parallelized framework code with hook functions. The programmer subsequently only needs to introduce serialized domain-specific functions that overrides these hook functions. The system has been implemented on a shared-memory multi-processor system.

Wirawan et al., [102] have introduced scalable and efficient parallelism to DNA sequence alignment problem. The authors have implemented a parallel DNA sequence alignment on the Cell Broadband Engine. The authors have also experimented with two parallelization methods - SIMD (Single Instruction Multiple Data) and wavefront. SIMD has been used within the SPEs while wavefront was used across the processor. The dataset used was artificial DNA sequences and the experiments were conducted with the IBM full system simulator.

Tan et al., [93] have proposed a parallel dynamic programming algorithm to solve problems of non-serial polyadic type. The authors have addressed this challenge by exploiting fine-grain parallelism and data locality. The authors have used multiple techniques such as helper threads (to read/write data in parallel with computations), parallel pipelining and tiling. The proposed algorithm achieves sub-linear speedup.

The targetted multi-core system is the IBM Cyclops64 supercomputer. However, IBM Cyclops64 supercomputer is an on-going project and no real machine has been built so far. Therefore, the authors have used the corresponding simulator C64-Simulator-FAST (Functionally Accurate Simulator Toolkit) for conducting all their experiments. FAST is designed for the purposes of architectural design verification and software development, and unlike the IBM Cell full system simulator, does not give accurate runtimes corresponding to what will be obtained in real hardware. The authors have experimented with varying tile size and noted the impact on the synchronization & communication overheads for various tile sizes.

Sadecki [82] has studied the possibilities of real implementations of a selected group of parallel dynamic programming algorithms. The experiments were conducted in the parallel multi-transputer SUPER NODE 1000 (SNODE) system and using the OCCAM programming language. The author used different configurations of connections between the computing elements such as master/slave and direct model. The author concludes that the proper choice of the system structure and the inter processor communication methods can considerably affect the efficiency of parallel computations and ultimately the speedup factor.

2.4.5 Parallel RNA Structure Prediction Algorithms

In this section, we highlight some of the relevant work done in the field of parallel RNA structure prediction algorithms. The objective is to understand some of the relevant parallelized RNA structure prediction algorithms.

Zhou et al., [109] proposed a new out-of-core distributed memory method for extended sequential RNA secondary structure prediction algorithms. The algorithm has several novel features such as redundant file scheme, I/O-reducing in-core buffer mechanism and dynamic load balancing. The algorithms utilize explicit file I/O operations to manage the data between in-core and out-of-core. In particular, it does not rely on the operating system (OS) to do the management using virtual memory; the authors argue that this will be suboptimal as the OS/VMM (Virtual Machine Monitor) is not aware of the application domain and the data dependencies.

The authors experimented with in-core distributed-memory parallelization and found that communication with neighboring processors is the key. In the redundant file scheme, the matrix stores the data from the rows and columns separately. This approach saves in-core memory when either the column or row is required. This approach is similar to creating indexes in databases servers. The authors conducted the experiments on a cluster of 16 Sun UltraSPARC IIIi nodes and have obtained good speedup. The authors have found that the size of in-memory buffer is critical for the efficiency of the parallel program. In particular, a large in-core memory buffer actually increases the number of memory access to cache misses.

Estrada et al., [32] have proposed a parallel framework compPknots that combines existing softwares Pknots-RE and Pknots-RG. The software predicts RNA secondary structures concurrently and automatically compares them with reference structures from database or literature. CompPknots is implemented using MPI

(Message Passing Interface) on a beowulf cluster to predict structures concurrently, thereby saving time and providing higher accuracies. The basic approach is still to run two existing algorithms concurrently on worker nodes, thereby exhibiting High Throughput Computing (HTC) where there is no data dependencies between the parallel jobs. The limitation of this approach is that the input data size that can be processed is still limited by one worker node's capabilities. The authors have used the software to predict 217 RNA structures from pseudobase database.

Nakaya et al., [67] have proposed a parallelized RNA secondary structure prediction method. The algorithm focuses on finding thermodynamically stable structures for single-stranded RNA molecules. The algorithm is based on a parallel combinatorial method that calculates the free energies of a molecule as the sum of the free energies of all the physically possible hydrogen bonds. The algorithm predicts many highly stable structures at once, although the structures are sub-optimal. This is contrary to most other algorithms that predict single optimal secondary structures. The core idea used in the algorithm is search tree pruning, with dynamic loading balancing across the processing elements in a parallel computer. The software has been implemented on CM-5.

Shapiro et al., [79] have proposed a RNA secondary structure prediction algorithm using Genetic Algorithm (GA) methodology. The algorithm has been implemented on a massively parallel supercomputer, MasPar MP-2, of type SIMD (Single Instruction Multiple Data) with 16,384 processors. Using this algorithm and setup, the authors have successfully predicted the existence of H-type pseudoknots in

several sequences. The results from these experiments match the phylogenetically supported tertiary structures of these sequences.

Shapiro et al., [81] have extended their previous work described in [79] to three different computer architectures. The algorithm was adapted to a 64 processor MIMD (Multiple Instruction Multiple Data) based SGI ORIGIN 2000 SMP system and a 512 processor MIMD CRAY T3E. Using the input sequences, the algorithm is initialized by generating a stem pool (consisting of either fully or partially zipped stems), which is then stored in all the individual processors. The parallel GA is initialized by stochastically picking stems from this stem pool. Evolution continues based on GA methods - selection, mutation and crossover individually across all the processors and in parallel. The results are measured using metrics such as runtime efficiency and prediction accuracy. Through this work, the authors have shown that it is possible to port existing algorithms to newer parallel computer architectures. They have also hinted at the optimizations that might be required to get better performances.

Liu and Schmidt [59] have proposed a parallel space-saving algorithm for aligning an RNA sequence to a SCFG using wavefront parallelization technique. The authors have implemented the algorithm on a PC cluster, a cluster of SMPs and recorded their speedups. On a 10-node PC cluster with each node being a dual-processor for a total of 20 processors the speedup was 16. Next on a 12-node SMP cluster, with each node being a quad-processor for a total of 48 processors, the speedup recorded was 36. The SMP is of hybrid type as the intra-node com-

munication is using shared memory while the inter-node communication is using MPI.

Fekete [34], has developed a novel parallel algorithm for RNA secondary structure prediction on distributed memory machines. The algorithm has been implemented on two distributed memory systems. The first system is an Intel iPSC/860 distributed memory parallel computer with 16 i860 processors. The second system is a Touchstone DELTA supercomputer consisting of an ensemble of processors organized as a two-dimensional mesh. The DELTA system is also a distributed memory parallel computer with maximum of 512 nodes. Subsequently, using the algorithm, the author has predicted and analyzed secondary structure of several long RNA sequences, including a complete HIV1 genome of sequence length 9229 nucleotides.

2.4.6 Parallel Computing Landscape

In this section, we highlight some of the important work done in the field of parallel computing. The objective is to understand some of the relevant academic work in the field of parallel computing. Asanovic et al., [7] review paper is an excellent summary of the landscape of the parallel computing research in 2006.

Petrini et al., [73] optimized Sweep3D on the Cell Broadband Engine and to their pleasant surprise observed several good performance trends such as high floating point performance reaching 64% of the theoretical peak in double precision,

and an overall performance speedup ranging from 4.5 times when compared with "heavy iron" processors such as IBM Power5, up to over 20 times with conventional processors. The authors have ported existing MPI-style software Sweep3D available in public domain using several handcrafted parallelization techniques. The authors also compare their results against other processors and also offer some architectural improvement suggestions. This paper is interesting as it shows that an existing MPI-style application registers better performance on a single multi-core CPU. This highlights the trend of using a single multi-core CPU for small and medium-sized problems.

Bader et al., [13] present a complexity model for designing algorithms on the Cell processor, and a systematic procedure for algorithm analysis. The execution time of the algorithm is estimated using computational complexity, memory access patterns (to and from SPUs in particular) and the complexity of branching instructions. The authors propose that the model and associated analysis procedure will likely simplify the algorithm design on the cell microprocessor and identify potential implementation bottlenecks. Subsequently, the authors have used this model to design an efficient implementation of list ranking. This paper is one of the early papers to analyze the cell architecture and show how to design efficient programs for the same.

Williams et al., [100] introduce a performance model for Cell and apply the same to several key scientific computing kernels such as dense matrix multiply, sparse matrix vector multiply, stencil computations and 1D/2D FFTs. The authors sub-

sequently validated the accuracy of their model by comparing their results against published hardware results, and also against the values from full system simulator. Additionally, the authors compared this performance benchmark against benchmarks from other HPC chips such as superscalar (AMD Opteron), VLIW (Intel Itanium 2), and vector (Cray X1E) architectures. The authors also propose modest micro-architectural modifications to increase the efficiency of double-precision calculations. Based on the results, the authors conclude that IBM Cell architecture is suitable for scientific computations in terms of both raw performance and power efficiency.

Beowulf cluster [85] architecture was designed during single-core processor era to amalgamate the processing capabilities of individual single-core computers into a single entity. The individual nodes were linked through ethernet. In this traditional configuration there was only inter-node communication and the performance limiter was the ethernet network. This led to the design and development of high-speed alternatives such a infiniband. The advent of multi-core system has introduced the problem of both intra-node communication using (perhaps) shared memory and inter-node using conventional MPI. It makes the design of cluster schedulers more complex in that they need to be aware of this two-tier architecture (nodes, cores) and schedule jobs accordingly. For example, two tasks that have data dependencies could benefit by being scheduled in the same node and pinned to different CPUs vs. being scheduled in different nodes.

Chai et al., [18], have designed a set of experiments to study the impact of multi-

core systems as individual nodes in a computer cluster like Beowulf clusters. The authors used popular benchmarks like HPL, NAMD and NAS as the applications to study. From the experiments, the authors found that on an average about 50% messages are communicated through slower intra-node communications. The author suggests that the trend indicates that intra-node communications must be optimized just like inter-node communications have been optimized earlier using faster communication networks. In addition, the authors observe that cache and memory contention may be potential bottlenecks and suggest that techniques such as data tiling can improve execution time. The authors suggest that in general newer applications should be multi-core aware. The results from this paper is very relevant to this research study and we have many similar observations. In general, the bottleneck in a multi-core is the communicating channels and the intermediate caches, rather than the endpoints (CPUs, Memory).

Paul and Meyer [72], observe that Amdahl's law [1] is based upon two assumptions - boundlessness and homogeneity - and argue that Amdahl's law does not apply to single-chip heterogeneous multi-processors (SCHM) architecture or even micro-architecture based systems. The authors have examined the implications of Amdahl's law on SCHMs and advocate that more research & design needs to be done and the processor performance should be viewed holistically and from a global perspective instead of local gains (such as one part of system, one task or program section) that could result in system-level slowdown. The authors infer that it will be more beneficial to invest time and effort in developing more

sophisticated system-aware schedulers rather than using off-the-shelf schedulers. Custom-designed schedulers will generally outperform generic schedulers, however the cost of developing an efficient one is the challenge. There is also an unknown factor on how individual systems will be clustered together in a real-world scenario. As an example, it is not uncommon for a computer cluster to contain nodes that are of different types. In such a case, the custom-scheduler may be of little use. We therefore argue that it may be beneficial to design a scheduling language that defines the entire hierarchy - network architecture, individual nodes features such as number of CPUs, their ISAs, number & size of various CPU caches. The scheduler should also know the data dependencies between parallel programs at various execution points. By this way, an intelligent scheduler will be able to better use the system resources.

Hill et al., [46] apply the historical Amdahl's law to hypothetical multicore chips of different configurations - symmetric cores, asymmetric cores and dynamically reconfigurable multi-cores. The authors have added a simple hardware model to fit the simple software model used by Amdahl in his famous argument. Based on their experiments they suggest that multicore designers should view the performance of a multicore chip entirely rather than focusing on core efficiencies. At the same time, they observe that obtaining optimal performance from single core is also important as there is likely to be sequential parts in any parallelized programs. They conclude by recommending that efforts be put into (automatically) parallelizing serial programs and also building more efficient parallelized hardware.

Chapter 3

Parallelizing PKNOTS

3.1 Introduction

Rivas & Eddy [76] proposed the first dynamic programming based algorithm that predicts optimal RNA secondary structure with pseudoknots and called it PKNOTS. The algorithm has a worst-case time complexity of $O(N^6)$ and worst-case space complexity of $O(N^4)$. The authors have implemented the algorithm in ANSI C programming language. The authors observed the program to run empirically in the complexities of $O(N^{6.8})$ for time and $O(N^{3.8})$ in space. The program scales above the theoretical complexity of the algorithm and the authors attributed this to the way the memory was allotted by the underlying operating system on the hardware used. The high complexity rates for the algorithm limited the program's RNA primary sequence input to be less than 150 nucleotides and

consequently the authors used the program to predict secondary structures for several small pseudo-knotted and non-pseudo-knotted RNAs.

The PKNOTS algorithm was introduced in the year 1999 while the modern high performance computing was still in its early stages of development. As discussed in the previous chapter, in the last decade, the computing landscape has seen the introduction of many new (& affordable) high performance computing architectures such as - faster processors, SMPs, cluster computing, grid computing, many-core architectures and more recently scalable cloud computing. The common pattern among most of these newer architectures is that they emphasize 'parallel computing'. On the other hand, the dynamic programing algorithm PKNOTS published in [76] evolves the final optimum secondary structure by identifying optimum secondary sub-structures recursively. By the nature of this definition, PKNOTS is a highly-recursive algorithm with deep data dependencies across multiple recursions. This property is generally considered be the bottleneck to parallelizing serial algorithms. Therefore, it is a very interesting problem to attempt to efficiently parallelize a dynamic programming algorithm and measure its performance on a variety of parallel hardware architectures and is one of the core contributions of this thesis.

In this chapter, we will explain the PKNOTS algorithm from the perspectives of algorithmic design and the data structures used in the program. We will refer the reader to the relevant sections of the publication [76] in order to avoid duplicating the material. Following this, we will explain the several parallelization techniques

that we have experimented with and implemented in different types of parallel computers. The techniques will be explained in detail in this chapter with the experimental results discussed in Chapter 5; the hardware used for experiments are detailed in the appendices.

3.2 Overview of PKNOTS

PKNOTS is the first single-sequence RNA secondary structure prediction algorithm that showed optimal RNA pseudoknot predictions can be made with polynomial time algorithms. The algorithm uses standard RNA folding thermodynamic parameters augmented by a few parameters describing the thermodynamic stability of pseudoknots. PKNOTS recursively searches for the optimum secondary structural motif(s) for sub-sequences of decreasing lengths and evolves the single optimal secondary structure for the whole input sequence at the end of the search process.

The algorithm makes a distinction between non-pseudoknot structures and pseudoknots structures that make up the RNA secondary structure. The algorithm refers to these as nested and non-nested structures and uses different search mechanisms. For the nested algorithm, the algorithm uses two triangular n x n matrices, called *vx* and *wx*. *vx(i,j)* is the score of the best folding between positions i and j, provided that i and j are paired to each other; whereas *wx(i,j)* is the score of the best folding between positions i and j regardless of whether i and j pairs to each or

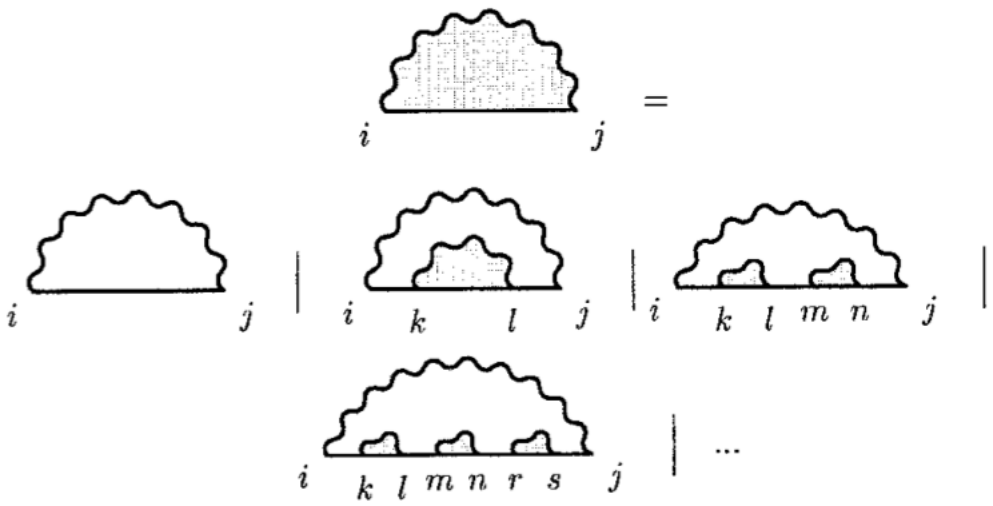


Figure 3.1: General recursion for vx in PKNOTS [76]

not. A nested dynamic programming algorithm fills the vx and wx matrices with appropriate numerical weights through a recursive calculation. The recursion for vx includes contributions due to - hairpins, bulges, internal loops, and multi-loops. The recursion is shown diagrammatically in Figure 4 of [76] and is reproduced in Figure 3.1.

The filling of matrix vx has been expressed mathematically in [76] and is reproduced here in Figure 3.2 along with initialization conditions in Figure 3.3. Each line gives the formal score for one of the several motif types and the optimal structure for this sub-sequence is the structure with the highest score. In particular, the first equation simply refers to a base pair, the second equation identifies hairpin loops, bulges, stems and internal loops. The rest of the higher-order equations are collected under the name of multi-loops, which have been observed to be much less frequent compared to their simpler counterparts.

$$vx(i,j) = \text{optimal}$$

$$\begin{cases} EIS^1(i,\ j) \\ EIS^2(i,j:k,l) + vx(k,l) \\ EIS^3(i,j:k,l:m,n) + vx(k,l) + vx(m,n) \\ EIS^4(i,j:k,l:m,n:r,s) + vx(k,l) \\ \quad +vx(m,n) + vx(r,s) \\ \mathcal{O}(5) \end{cases}$$

$$[\forall k,l,m,n,r,s, \quad i \leqslant k \leqslant l \leqslant m \leqslant n \leqslant r \leqslant s \leqslant j]$$

Figure 3.2: Mathematical formulation of general recursion for vx in PKNOTS [76]

The initialization conditions are:

$$vx(i,i) = +\infty, \quad \forall i \quad 1 \leqslant i \leqslant N$$

Figure 3.3: Initialization condition for general recursion of vx in PKNOTS [76]

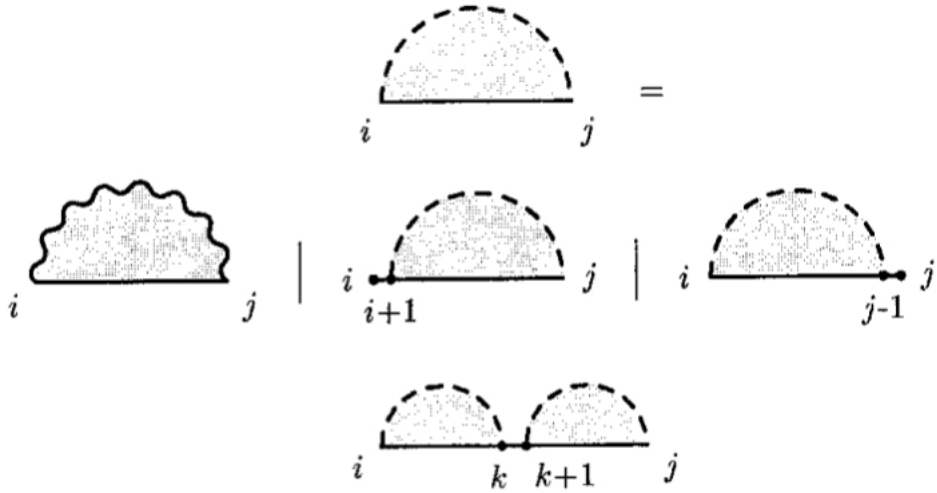


Figure 3.4: General recursion for wx in PKNOTS [76]

$$wx(i,j) = \text{optimal}\begin{cases} P + vx(i,j) &] \text{ paired} \\ \left.\begin{array}{l} Q + wx(i+1,j) \\ Q + wx(i,j-1) \end{array}\right] & \text{single-stranded} \\ wx(i,k) + wx(k+1,j) \quad [\forall k, \quad i \leqslant k \leqslant j]. &] \text{ bifurcation} \end{cases}$$

Figure 3.5: Mathematical formulation of general recursion for wx in PKNOTS [76]

The recursive relations used to fill the wx matrix identifies the remaining types of secondary structural motifs - single-stranded nucleotides, external pairs, and bifurcations. The actual recursive process is easier to understand using a diagram and is given in Figure 7 in [76] and is reproduced here in Figure 3.4. The filling of this matrix wx is given mathematically by Equation 5 and initialization condition in Equation 6 from [76] and are reproduced here in Figures 3.5 and 3.6.

With the initialization condition:

$$wx(i,i) = 0, \quad \forall i \quad 1 \leqslant i \leqslant N$$

Figure 3.6: Initialization condition for general recursion of wx in PKNOTS [76]

Pseudoknots are non-nested configurations and cannot be described by using only *wx* and *vx* matrices. In order to handle pseudoknots, PKNOTS introduces two new gap matrices in addition to the existing non-gap matrices. Pseudoknots are described visually in Figure 8 in [76] and based on this can be described using two gap matrices with complementary holes. The gap matrices are of type one-hole, which means the pseudoknots are restricted to be of class simple pseudoknots. However, it is possible to use multiple one-hole gap matrices to model generalized pseudoknots.

The interesting aspect of using one-hole gap matrix is that non-gap matrix *vx* and *wx* can be represented as a special case of using gap matrix with no-hole. They are shown visually in Figure 10 and Figure 11 in [76] as well. Corresponding Equation 8 and Equation 9 in [76] show the mathematical formulation. In total, 4 matrices - *VHX*, *WHX*, *YHX*, *ZHX* are used to model the relationship between bounding base pairs of a pseudoknot. The details of the relationship are shown as Table 1 in [76]. This brings the total number of matrices to 6 in PKNOTS. PKNOTS uses the turner thermodynamic information for non-nested secondary structures and estimates the corresponding thermodynamic parameters for pseudoknot (as none was available at the time the paper was written). These are again shown in Table 2 and Table 3 in [76]. Mathematical formulations for the four gap matrices are described and discussed in appendices section of [76]. Finally, the algorithm is also able to handle co-axial stacking and dangle type secondary structural motifs. Figure 3.7 shows from a mathematical perspective all the types of motifs that are

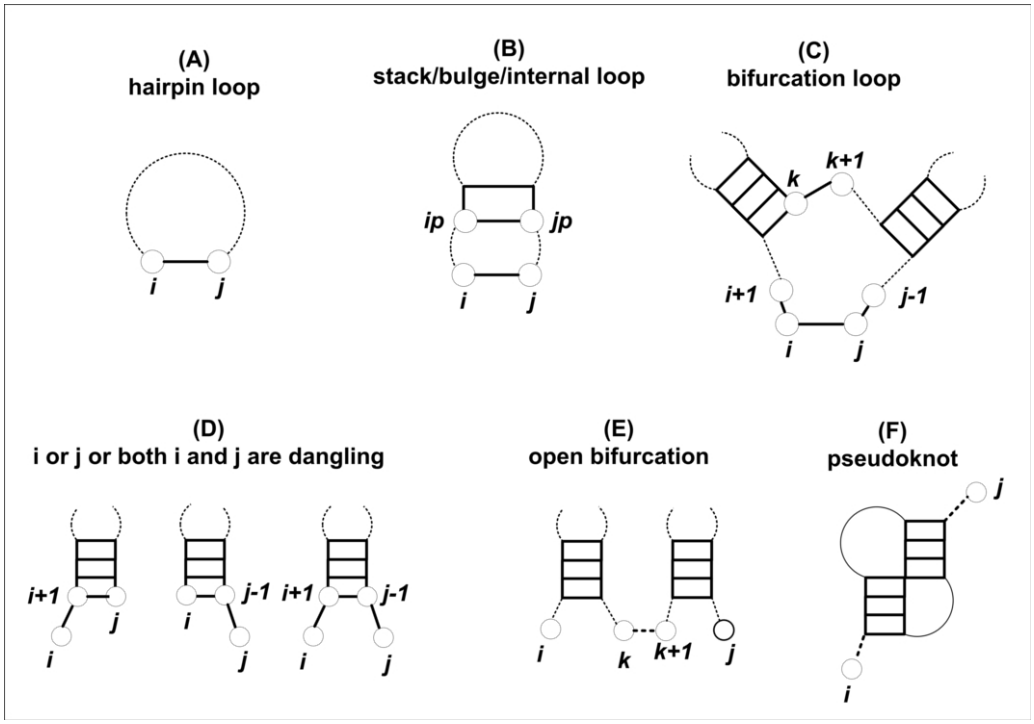


Figure 3.7: Motif types searched by PKNOTS algorithm

searched by the PKNOTS algorithm.

3.3 Analyzing PKNOTS

There are three key reasons why we choose to parallelize PKNOTS against other alternative algorithms. First, PKNOTS was the first dynamic programming algorithm to be able to predict secondary structures including pseudoknots (albeit not all types) using a single RNA primary sequence. This motivated us to work with a novel algorithm. Second, the time and space complexities for PKNOTS is on a higher-end. Therefore, introducing efficient parallelism from today's parallel computing architectures to the algorithm will likely bring the biggest gains. Third, PKNOTS implementation is available freely in source code form, unlike

several other programs that are only provided as web applications. This allows us to tweak & port the algorithm to multiple parallel architectures and subsequently measure their performances as well. The following paragraphs describe the several generic parallelization techniques that we have experimented with. The architecture-specific parallelization-tuning optimizations and results from the experiments using them are discussed in Chapter 5.

As described in the previous section, PKNOTS contains 6 core matrices - 2 for non-pseudoknots and 4 for pseudoknots - that are used to predict secondary structures. In the reference software implementation, there is a corresponding sub-routine for each of these matrices. The sub-routines are known as *FillVX, FillWX, FillVHX, FillWHX, FillYHX and FillZHX*. The functions for pseudoknots run in time complexities of $O(N^6)$ while the functions for non-pseudoknots run in time complexities of $O(N^5)$. In addition to these core matrices, the software program uses other supporting matrices as well. Figure 3.8 shows the pseudocode for core part of PKNOTS algorithm while Figure 3.9 shows the program flow of this matrix filling routine.

From the pseudocode in Figure 3.8, it can be seen that the matrices are highly interdependent on each other and moreover the dependency is non-serial polyadic. Non-serial polyadic property means the value to fill a particular matrix cell is dependent not just on the value from the immediate previous recursion but on historical value(s) as well. Figure 3.10 shows in matrix form the data dependencies between the various matrices as well; data dependencies are from row to column.

Pseudo code for subroutine FillMtx

```
for j = 0 to seqlen
     for d = mind to j+1
        FillVP(vx, vp, j, d);
        FillWX (wx, vx, whx, yhx, j, d)
        FillWBX(wbx, vx, whx, yhx, j, d)
         for d1 = 0 to d+1
              for d2 = 0 to d-d1-1
                  FillVHX(whx, vhx, j, d, d1, d2)
                  FillZHX(wbx, vx, whx, vhx, zhx, j, d, d1, d2)
                  FillYHX(wbx, vx, whx, vhx, yhx, j, d, d1, d2)
                  FillWHX(wbx, vx, whx, vhx, zhx, yhx, j, d, d1, d2)
              end for
         end for
     end for
end for
```

Figure 3.8: Pseudocode for matrix filling routine in PKNOTS algorithm

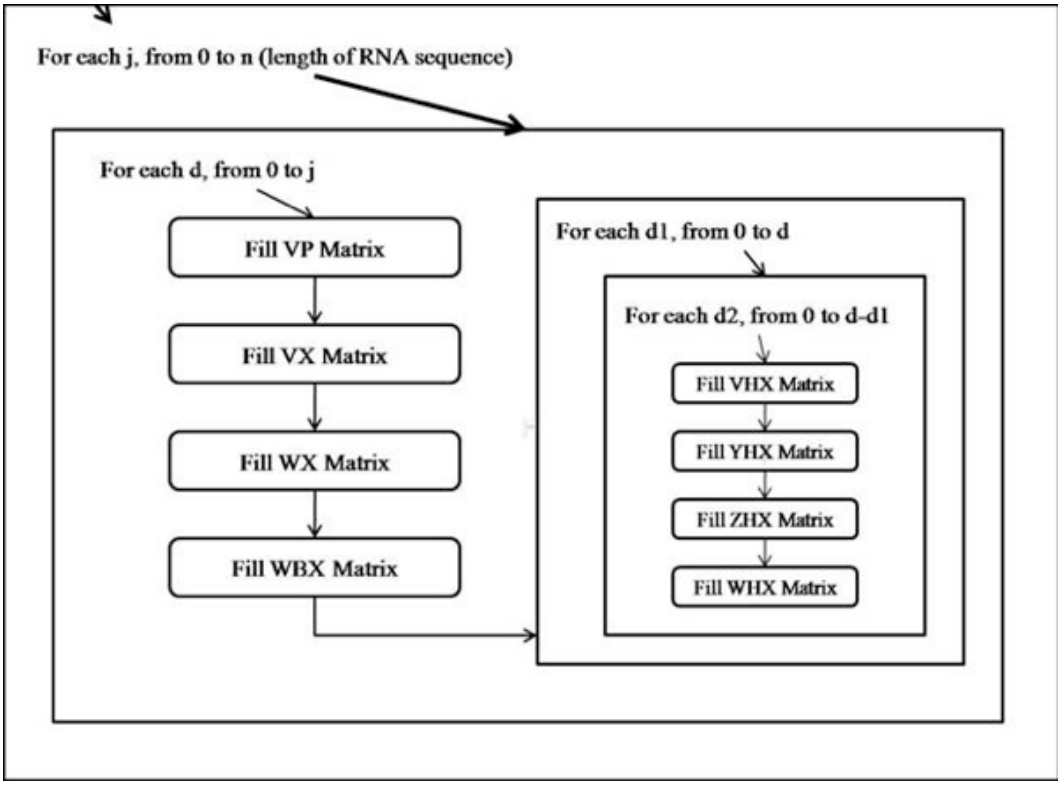


Figure 3.9: Program flow of the matrix filling routine in PKNOTS algorithm

	VP	VX	WX	WBX	VHX	ZHX	YHX	WHX
VP	O	O						
VX	O	O		O		O	O	O
WX		O	O				O	O
WBX		O		O			O	O
VHX					O			O
ZHX		O		O	O	O		O
YHX		O		O	O		O	O
WHX		O		O	O	O	O	O

Figure 3.10: Data dependencies across matrices in PKNOTS algorithm

Every matrix filling function is also dependent on thermodynamic parameters that govern the pairing of bases in the structure. The thermodynamic parameters are provided as a two dimensional matrix, named ICFG, that is of size 4276×4276 elements. The four gap matrices are four-dimensional while the other three matrices are two-dimensional yielding the maximum algorithmic space complexity of $O(n^4)$.

3.4 Parallelizing PKNOTS

In this section we will describe our efforts to introduce parallelization into the PKNOTS algorithm. We begin by analyzing the scalar implementation to identify

suitable locations in the algorithms for introducing parallelization. Following this we will describe the three-parallelization methods that we have designed. We attempted various strategies and exploited several relationships among the code graphs and data paths of the PKNOTS scalar implementation. We describe all these in the following subsections.

3.4.1 Measuring PKNOTS's Performance

Amdahl's law *Amdahl1967* states that the amount of speedup achievable is bounded by $(1/((1-p)+p/s)$ where 'p' is the percentage of the original code where a speedup of 's' is obtained. Therefore, to obtain an upper bound for parallelism we set out to identify the section of the program where most of the time is spent. A timing analysis was performed on the scalar implementation, to clock each matrix filling function. The result of the timing analysis is shown in Figure 3.11. From the figure, it is clear that a bottleneck exists at the function that fills the *WHX* matrix. This is both because the function contains large amounts of computations and is located four levels deep in the nested configuration of the program. Therefore, the function is invoked for every variation of the four loops, whose variables are denoted by *j*, *d*, *d1* and *d2* in Figure 3.11. For that reason, the extremely low weights of the functions for *VHX*, *ZHX*, and *YHX* is not considered for parallelization efforts as the performance gains are likely to be insignificant. Therefore, the parallelization efforts focus solely on this function which fills the *WHX* matrix, simply because of the *90%* contribution of this function towards the program's

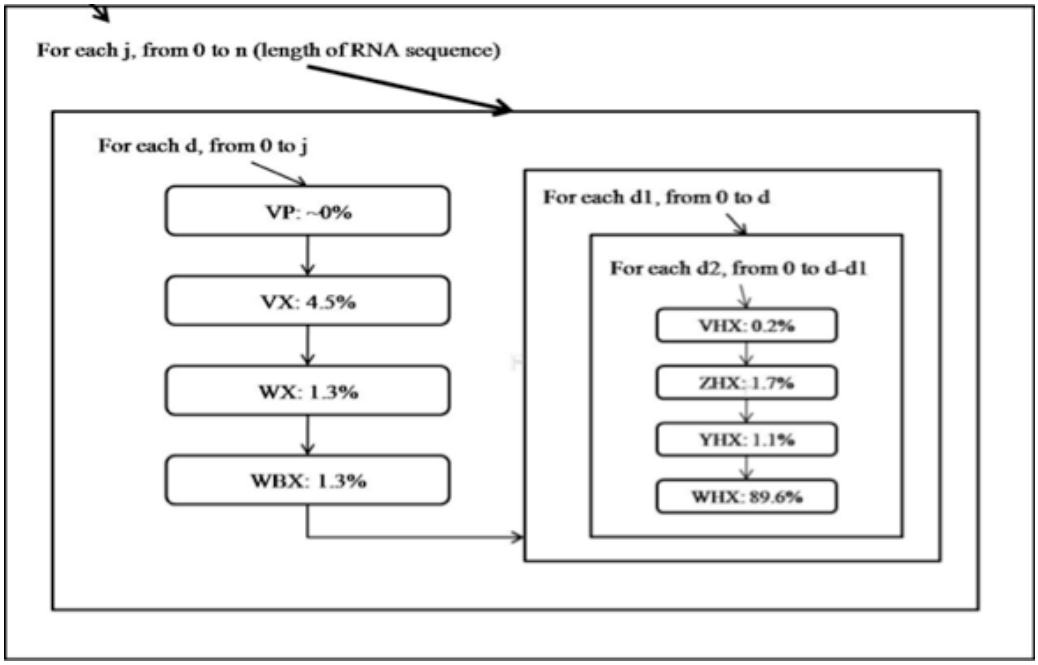


Figure 3.11: Timing Analysis of PKNOTS Algorithm

runtime.

Next, we seek to understand the structure of the *WHX* code block. *WHX* consists of six major blocks of work. Each major block contains a large amount of comparisons between values from other matrices. A point to note is that there is no data dependency between the individual blocks. Figure 3.12 shows the WHX function with each block of work named WHXn where 'n' is in the range from 1-6. Each block is essentially a loop and the number of borders indicates the level of nesting. Note WHX1 does not contain loops and therefore in all subsequent parallelization WHX1 is left untouched.

In addition to the individual WHXn loops (or work units) in the WHX matrix-filling function being independent of each other, the loops themselves are each independent across iterations. We use this to our advantage by performing SIMD-

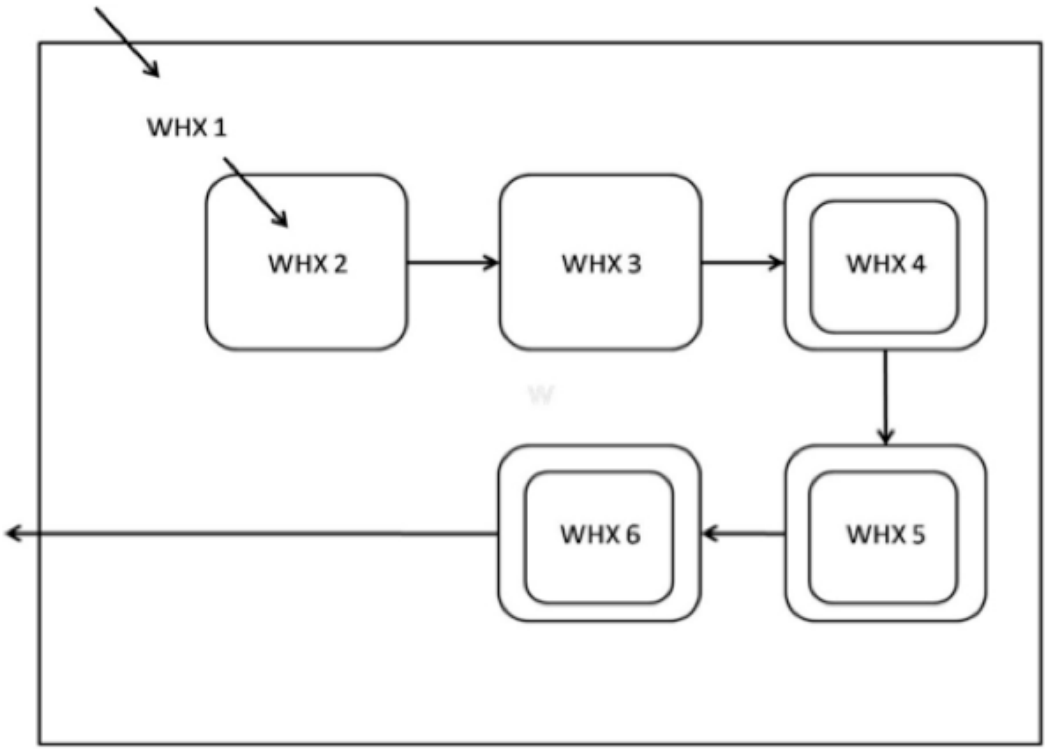


Figure 3.12: WHX layout in the PKNOTS Algorithm

style loop parallelization. We begin by using a CE (Consumer Electronics) class parallelized computer, namely the Sony PS3, to do small-scale experiments prior to doing large-scale experiments with parallelization across multiple parallelized architectures. Techniques we describe in this chapter have largely remained the same across the different architectures, with required changes to optimize them to suit the various system specifics.

3.4.2 Code Parallelization (C-Par)

In the C-Par model, we parallelize the algorithm by splitting it into WHX processing and non-WHX processing. The WHX processing is again parallelized into six

routines, each handling one of six WHX blocks WHX1 to WHX6. In this model, data independence between the individual WHX blocks is exploited. Each block of work for filling WHX is now performed separately and the execution of each block is performed in parallel.

In order to test the feasibility of this model, we ported the PKNOTS algorithm to the IBM Cell [112] platform and tested it out using the Sony PS3. The effort of porting required making sizable changes to the source code in order to accommodate the platform intricacies. However, since the programming language was common between the platforms, we didn't have to recode the algorithm in another programming language.

The IBM Cell architecture will be explained in much detail in Appendix C and we will provide a very brief overview here for the reader to understand the sections below. IBM Cell is a heterogeneous multi-core processor comprising of two different types of processing elements each with its own Instruction Set Architecture (ISA). The processing cores are PPE (Power Processing Element) and SPE (Synergistic Processing Elements). There is 1 PPE and up to 8 PPEs in a single cell processor; Sony PS3 has only 6 SPEs. The PPE is capable of running an executive process like the OS while the SPEs acts as specialized co-processors.

In this architecture, the PPE's primary task is to run the non-WHX routines while managing the synchronization between the SPEs. The strategy is to run six SPEs, each handling one unique block of work, in essence running concurrently all the work 6 blocks of WHX, WHX1 to WHX6. Figure 3.13 illustrates this model

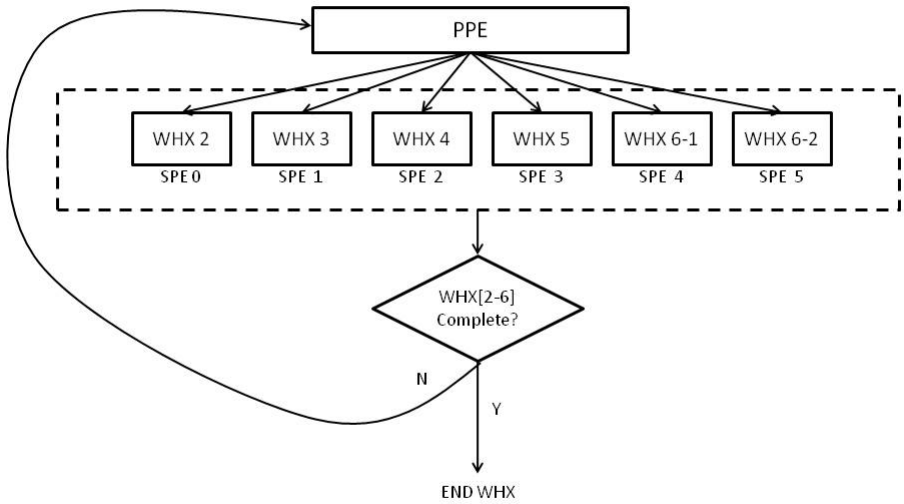


Figure 3.13: C-Par model of PKNOTS on Sony PS3

on the Sony PS3 platform. Based on timing analysis we conducted, the C-Par implementation didn't make effective use of the available computational power in the SPEs due to different runtimes of various WHX blocks (i.e., number of loop iterations) in matrix-filling functions and thus causing synchronization delays. In our next implementation, we aim to balance out the workload among the available SPEs.

3.4.3 Data Parallelization (D-Par)

In this model, each SPU executes all the six functions sequentially for the same piece of input. This mitigates some effects of slowdown in C-Par model due to the non-constant number of iterations performed in each WHXn block. SPE contexts,

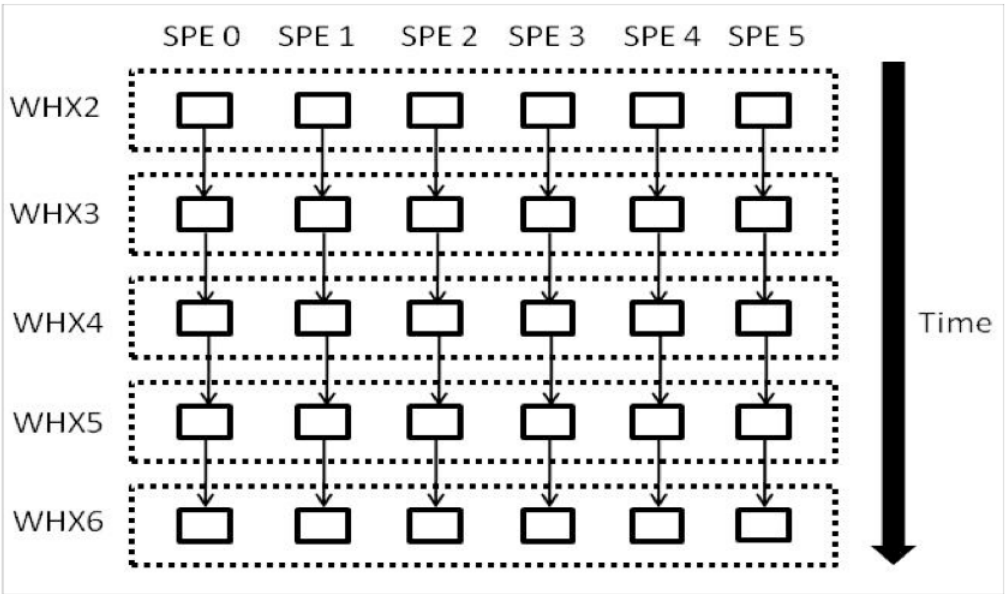


Figure 3.14: D-Par model of PKNOTS on Sony PS3

an abstraction of SPE containing the code and data, was used in the implementation. Figure 3.14 shows the D-Par implementation in a graphical way.

Through our experimental investigations, we found that SPE context creation and swapping is expensive as the functions are nested in loops and SPE contexts are swapped repeatedly in this finer-grain parallelization model. The overheads of setting up the parallel environment is amplified by the time complexity of the algorithm as well. This created the issue of SPU initialization as different function codes needs to be swapped in and out of the SPU for each and every piece of data at various stages of the WHX2-6 functions. We eliminated the inefficiency associated with SPU initialization, by using a single binary image. The single binary image contains all the WHX functions and using input parameters, the scheduling task will let the worker task know which functions to execute.

3.4.4 Hybrid Parallelization (H-Par)

Both the C-Par and D-Par models have their own unique strengths (and weaknesses). Although, they differ on what gets executed on the SPE and when, they both are alike in that algorithmic computation is always done on the SPEs and the task in PPE is simply synchronizing the different SPE tasks. We wanted to find out the trade offs between communication costs vs. computation gains of executing tasks locally vs. remotely. We call this the H-Par model. H-Par model is essentially a runtime tuning mechanism that decides on the locality of execution, i.e., whether to run on PPE or to distribute across SPEs. We use the input RNA sequence length as a threshold value and whenever it is below a certain length, the scheduler will run the code in the PPE and avoids invoking the SPEs, as the communication and synchronization costs are more than the computational gain achieved by executing the functions in the SPEs. For longer RNAs, SPEs are chosen for computation workloads. Figure 3.15 shows the flow chart of this implementation strategy.

3.4.5 Preliminary Results

The parallelization methods and strategies that we explored in the previous sections give us valuable knowledge into parallelizing an existing dynamic programming based algorithm called PKNOTS. We also tested the various methods with a small dataset and Figure 3.16 shows the performance differences between the

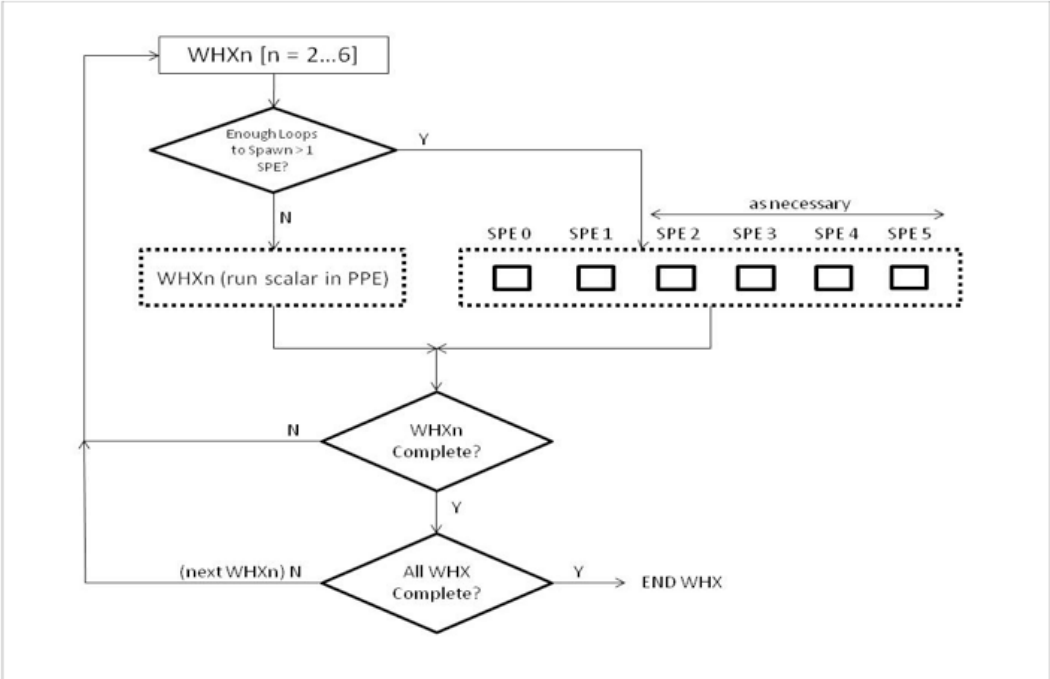


Figure 3.15: H-Par flow chart of PKNOTS on Sony PS3

various methods. It can be seen that each successive method has better performance with H-Par, which incorporates intelligent scheduling, having the best performance. This clearly sets the motivation to design a new algorithm for RNA secondary structure prediction and also do large-scale experiments with PKNOTS across three parallel architectures.

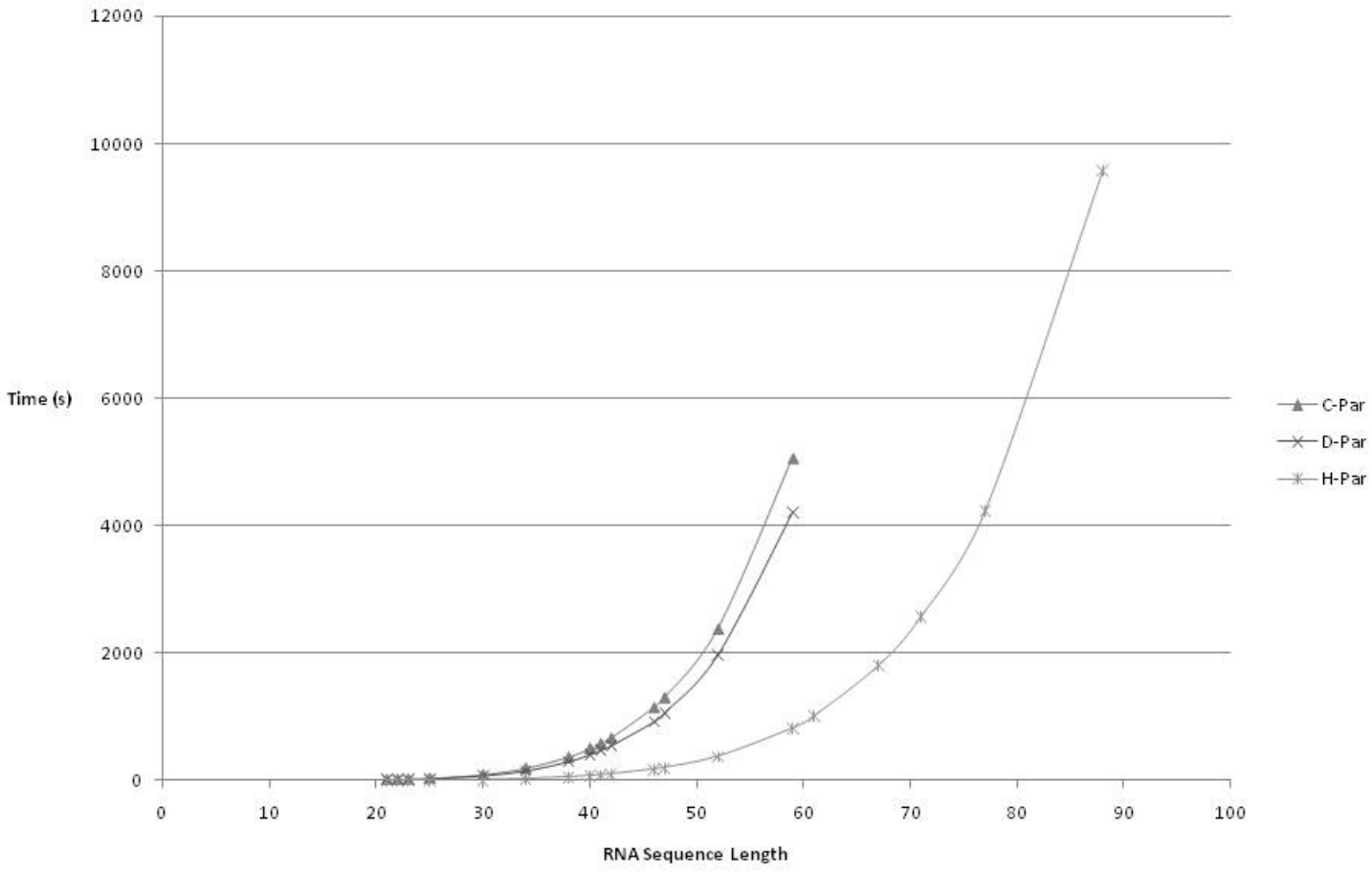


Figure 3.16: Preliminary results with PKNOTS on Sony PS3

Chapter 4

MARSs

4.1 Introduction

In this chapter, we introduce and describe in detail a novel algorithm that we are proposing for the prediction of RNA secondary structures. The algorithm is called "A Matrix Algorithm for RNA Secondary Structure Prediction" and is abbreviated as MARSs. MARSs is capable of predicting all of the currently known structural motifs of a RNA secondary structure. MARSs is shown to have polynomial time and space complexities of $O(n^3)$ and $O(n^2)$ respectively. Following are the key salient and distinguishing features of MARSs.

MARSs is a top-down algorithm. This means that the algorithm first predicts secondary structure motifs at the macro or whole-sequence level and then fills in the gaps created. This is in contrast to Dynamic Programming

(DP) based algorithms such as PKNOTS, which are bottom-up algorithms.

MARSs evolves secondary structures. MARSs does not perform a library-based search for predicting secondary structural motifs. Instead, it evolves the complete secondary structures as a whole.

MARSs is a non-recursive algorithm. MARSs algorithm has been designed specifically for higher performance and one of the design objective is to easily parallelize it on multi-CPU architecture. To achieve this, the algorithm uses a non-recursive design unlike DP based algorithms such as PKNOTS.

MARSs can predict alternate structures. DP based algorithms by definition grow and evolve one optimum secondary structure. In contrast, MARSs is capable of predicting a set of high-quality structures for a given sequence. By this way, for sequences with unknown secondary structures alternate output from MARSs can be considered.

MARSs algorithm is targeted at High Performance Computing (HPC) architectures and seeks to eliminate the performance shortcomings found in traditional algorithms based on Dynamic Programming (DP). As such, we believe that MARSs will perform equally well in HPC architectures from multi-core CPU based SMPs to cloud-based HPC architectures. The implementation features auto-scaling and detects the number of parallel cores available in the system during program startup. At the same time, the algorithm performs equally well in single core CPUs with no reduction in the prediction accuracy.

4.2 RNA Secondary Structure

RNA primary structure is a linear sequence comprising of RNA nucleotides - Adenine(A), Cytosine(C), Guanine(G) or Uracil(U) as alphabets. The string of nucleotides folds upon itself and in the process forms hydrogen bonds amongst themselves. When two nucleotides (or bases) forms a bond, the bases are collectively known as the base pair. The base pair is known to be in a more stable state (compared to the primary sequence) due to the reduction in free energy. Therefore, the more the base pairs are formed, the more stable is the resultant secondary structure. The secondary structures that are naturally possible have been documented by the biologists and are listed below

- Bases of the same kind do not bond with each other. For example, Adenine(A) cannot bond with another Adenine(A)
- Two successive nucleotides in the primary sequence cannot bond with each other again
- The allowed base pairs are $A - U$, $C - G$, $A - C$ and $G - U$
    - Energy reduction of $A - U$ and $C - G$ bonds are approximately the same (A)
    - Energy reduction of $A - C$ and $G - U$ bonds are approximately the same (B)
    - Energy reduction of case 'A' above is more than case 'B'

- The structure that has larger energy reduction is more stable and preferable.

As the RNA alphabet size is only 4, the number of possible combinations among nucleotides is quite small. Therefore, the challenge is to determine the optimal configuration of the RNA secondary structure that is likely to produce the more stable secondary structure.

4.3 Algorithm Initialization

Step 1 of the MARSs algorithm is to initialize the secondary structure prediction process. We do this by capturing the biological rules discussed in the previous section in two data structures. As the name of the algorithm indicates, these data structures are of matrix type. The two matrices are called as *Base-Pair Matrix (BPM)* and *Affinity Matrix (AM)*.

The *Base-Pair Matrix (BPM)* is a static matrix such that the contents of the matrix is fixed and does not change with different input RNA primary sequences. The matrix stores the bond strengths among the nucleotides and can contain either integer or floating-point values. The values can be updated before each run of the MARSs algorithm and the algorithm auto-updates the prediction of the secondary structure using the new values. By this way, MARSs can be used to predict RNA sequences from different organisms with ease, such as plants and animals, which might use different bonding energies. Table 4.1 shows the simplified version of a *Base-Pair Matrix* using relative bond strengths; the larger the number the stronger

Table 4.1: Base-Pair Matrix

*	A	C	G	U
A	0	1	0	2
C	1	0	2	0
G	0	2	0	1
U	2	0	1	0

the bond. Therefore, in this case the total energy of the predicted structure is *energy maximization* instead of the traditional *energy minimization*. It is trivial to use negative energies and invert the corresponding value comparison mathematical operators in the prediction algorithm to mimic the energy minimization behavior in conventional algorithms. The bond strengths can then be seen from the perspective of energy reduction as well. The bonding strengths in particular do not consider the effects of nearest neighbor nucleotides or other constraints like being part of a loop or stem.

As can be seen in Table 4.1, *Base-Pair Matrix (BPM)* is always a 4 x 4 matrix irrespective of the sequence length. In the above example, we have used a scoring value '2' to represent Watson-Crick (A-U and C-G) base pairs, while Hoogsteen (A-C) and Wobble (G-U) are given a scoring value of '1'. Base pairing of the same bases are given a score of '0' that means that bonding between them is not possible. The chosen scoring model highlights the fact that the Watson-Crick base pairs are stronger than both Hoogsteen and Wobble base pairs and this multi-level scoring scheme guides the MARSs algorithm to predict stronger bonds wherever possible.

Table 4.2: Affinity Matrix

*	G	G	U	U	A	G	U	U	C	C
G	0	0	1	1	0	0	1	1	2	2
G	0	0	0	1	0	0	1	1	2	2
U	1	0	0	0	2	1	0	0	0	0
U	1	0	0	0	0	1	0	0	0	0
A	0	0	2	0	0	0	2	2	1	2
G	0	0	1	1	0	0	0	1	2	2
U	1	1	0	0	2	0	0	0	0	0
U	1	0	0	0	2	1	0	0	0	0
C	2	2	0	0	1	2	0	0	0	0
C	2	2	0	0	1	2	0	0	0	0

The second matrix that is required for MARSs to predict RNA secondary structures is called *Affinity Matrix*. Unlike the *Base-Pair Matrix*, *Affinity Matrix* is constructed on a per-sequence basis. For every RNA primary sequence, *Affinity Matrix* is constructed using the raw bonding values from *Base-Pair Matrix*. Therefore, the size of the matrix is a square of the length of the primary sequence. Next, we apply the 'neighbor no bonding' rule as explained in Section 4.2. This is achieved by making 3 diagonal columns zeros - that match the rule *(n,n), (n-1,n) and (n,n+1)* where *'n'* is the nucleotide position. Table 4.2 shows the affinity matrix using a hypothetical sequence of 10 nucleotides, whose primary sequence is "GGUUAGUUCC".

Our hypothesis is that the *Affinity Matrix* contains all the possible secondary structures predictions for the primary sequence. By selectively traversing along the *Affinity Matrix* we can extract the different secondary structures at much less computation cost than dynamic programming based prediction algorithms.

4.4 Level 1 Folding

In general, a RNA secondary structure is formed when the primary structure folds upon itself. We refer to this as *Level 1* folding. Every nucleotide can be a folding point and therefore the number of folding points is equal to the length of the input sequence. The number of folding points also grows linearly with the length of different sequences. Let us represent the length of a given sequence using variable *'n'*. We now need to identify the number of potential bases these folding points can bond, to form the first base pair for this fold. As a constraint, we also need to factor in the neighbors-no-bonding rule that states that two naturally occurring neighbor nucleotides cannot form a base pair through hydrogen bonding.

Let us visualize the process of computing the total number of *Level 1* folds possible for a given RNA primary sequence. The visualization process consists of two stages. In the first stage, let us suppose that any nucleotide (folding point) can bond with any other single nucleotide to form the first base pair. In the second stage, we apply the neighbors-no-bonding rule and eliminate the impossible bond pairs and count the remaining possible base pairs. Using an anonymous sequence of four nucleotides the Figure 4.1 shows this process. using this visualization as an example, we mathematically represent it to compute the total number of *Level 1* folds for a sequence of any given length.

In *Stage 1*, for a sequence containing four nucleotides, the first nucleotide has 3 bases to bond with. The second nucleotide has 2 bases to bond with and the third

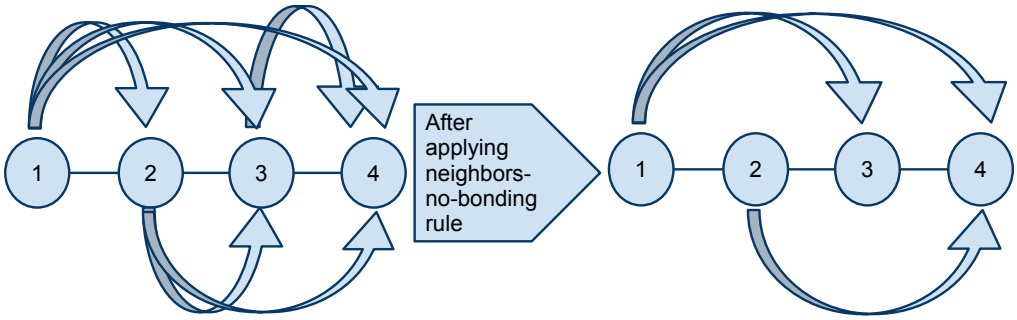


Figure 4.1: MARSs Folding Points

nucleotide has 1 base to bond with. Hence, the max number of folding points in *Stage 1* for any sequence of 4 nucleotides = 3 + 2 + 1 = 6. This can be written as *(n-1) + (n-2) + (n-3)* or written as a summation equation as $\sum_{x=1}^{n-1}(n-x)$. Simplifying this to a mathematical form we get equation $n(n-1)/2$.

In *Stage 2*, we apply the neighbors-no-bonding and this reduces the maximum number of folding points from 6 to 3 for a sequence of 4 nucleotides. This reduction in the number of base pairs can be represented in two ways - either as the number of naturally occurring bonds or as the number of remaining number of bases from the primary sequence after subtracting the number of bases to skip-over. Representing this in a mathematical form we get equation $n(n-1)/2-(n-1)/1$. Simplifying this further we get equation $(n-1)(n-2)/2$.

This represents the total number of folding points that is possible for any sequence of length *'n'*. For example, a sequence of length 8 nucleotides can fold upon itself in 21 different ways. The challenge is to determine which of these 21 folds will produce the most optimal secondary structure as there is only one optimal sec-

ondary structure for a given RNA sequence. The above stage can be generalized to accommodate organism-specific rules for skipping a minimum number of nucleotides for establishing the folding pair. The above mathematical representation can therefore be generalized to representation $n(n-1)/2-(n-m)/1$ where *'m'* is the number of nucleotides to skip and *'m'* can be any value between *'1'* and *(n - 2)*.

For each of these folds, we will use the nucleotide bonding values from *Affinity Matrix* to determine the total number of base pairs (and therefore total energy) for each of the folded sequences. As an example, referring to the hypothetical sequence in the *Affinity Matrix* in Table 4.2 and using nucleotides *(U,A)* at positions *(2,4)* as folding points the Figure 4.2 shows the secondary structure for this sequence. It can be observed that there are two bonds - *(U,A)* and *(G,U)* - at locations *(2,4)* and *(0,6)*. The bond between nucleotides *(2,4)* is of type *Watson-Creek* while the bond between *(0,6)* is of type *Wobble*. This folding point at *(2,4)* has created a hairpin loop comprising of a single nucleotide *(U)* at position *(3)*. It has also created a symmetric internal loop between nucleotides *(0-2)* and *(4-6)* and a free dangling end between nucleotides *(7-9)*.

It can also be observed from Figure 4.2 that MARSs has proposed a bond *(G,U)* between nucleotides *(0,6)* instead of nucleotides *(1,6)* although both of the bonds will be exactly the same. We call this type of prediction as symmetric-fold and will be explained in more detail in the next section along with a second type of prediction called as asymmetric-fold.

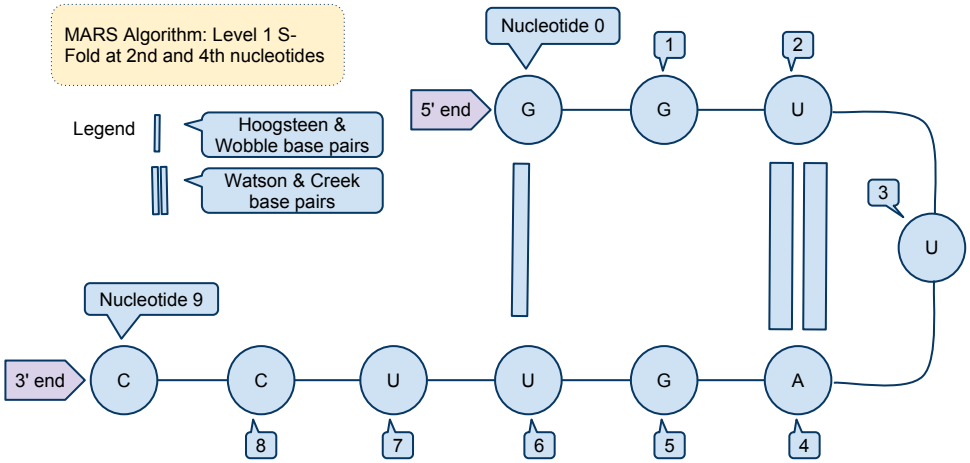


Figure 4.2: MARSs Level 1 Symmetrical Folding

4.5 Symmetric Folding (S-Fold)

There are several well-known secondary structural motifs - *hairpin loops, stems, bulge, internal loops (symmetrical & asymmetrical), free dangling ends, junction and pseudoknots (simple and generic).* MARSs uses a systematic approach to discover secondary structural motifs and does not employ a library-based search method.

The first step in the process of evolving a secondary structure is fixing the folding point. This has been described in detail in the previous section. Once the folding point has been fixed, the algorithm processes the nucleotides on the side opposite from where the loop has been introduced. The nucleotides at the folding point may bond depending on the type of nucleotides on either side. There are two outcomes.

The first outcome is that a bond is predicted. In this case, the search continues by

moving one nucleotide on either side of the bond. If the original folding point is at positions *(n,m)* then the search will continue from the indices that is derived by subtracting 1 from one of the indices and adding 1 to the other index - *(n-1,m+1)*. In the above example, if the folding point was at $n = 2$ and $m = 4$ the new indices will be $n - 1 = 1$ and $m + 1 = 5$. Now, the search continues using the new indices as the base $n = n - 1$ and $m = m + 1$. Using the same hypothetical sequence as an example, we can see that the nucleotides at positions *(1,5)* cannot form a bond as they both are the same molecule - Guanine(G). In this situation the output is the same as the first outcome that is possible with the original bonding pair. The second outcome is that a bond is not feasible at the folding point.

The scenario in the previous paragraph gives us two choices. First, move both sides of the RNA strand by the same number of nucleotides, one in this case. Second, move the two sides of the RNA strand using different number of nucleotides. We call the first type as *Symmetrical Fold* and the second type as *Asymmetrical Fold*. *Asymmetrical Fold* can further be sub-classified and will therefore be explained in detail in the next section. *Symmetrical fold* allows the prediction of the third type of secondary structural motif - internal loops (symmetrical) - in addition to the stem and hairpin loop predicted as part of the first folding point/pair selection. This is depicted in Figure 4.2.

4.6 Asymmetric Folding (A-Fold)

Symmetric Fold as the name suggests, is capable of predicting secondary structural motifs that have symmetry across the base pairs such as stems, hairpin loops and internal loops (symmetric). At the same time, a RNA secondary structure may comprise of other types of structural motifs that are asymmetrical by nature such as bulges and asymmetrical internal loops. In order to equip MARSs with the capability to predict asymmetrical motifs we introduce a second type of folding called *Asymmetric Fold* or *A-Fold* in short in this section.

Let us start by describing what exactly is meant by asymmetry from the perspective of a RNA secondary structure prediction. The bonding process starts by arranging the nucleotides so that they are facing each other, either when the primary folding base pair is formed or after a previous base-pair prediction. After this, if a base pair is predicted between nucleotides that are opposite to each other, then there are either *0* or equal number of nucleotides between base pairs. For example, in Figure 4.2 the base pair between nucleotides *(0,6)* is the newly predicted one and is two nucleotides (one on each side) from the previous prediction at nucleotides *(2,4)*. The free nucleotides at positions *(1,5)* form a symmetrical internal loop. On the other hand, if the new base pair was formed between nucleotides *(1,6)* instead of *(0,6)* then there would be only one free nucleotide at location *(5)*. This would have resulted in bulge rather than a symmetrical internal loop. A-Fold predicts these kinds of structures.

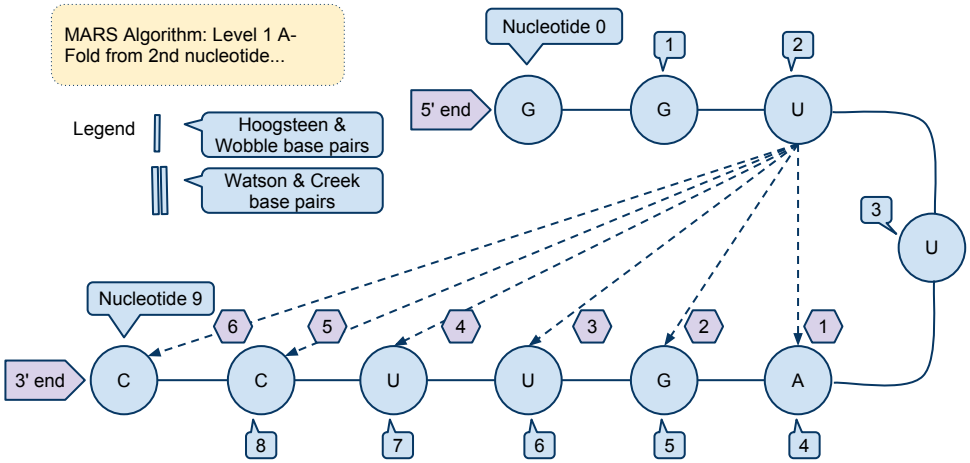


Figure 4.3: MARSs Level 1 Asymmetrical Folding types - 1

Asymmetrical structures can have more free nucleotides either on the 5' or 3' side of the primary sequence. Any prediction algorithm must cater for this possibility. MARSs achieves this by scanning for potential base pairs using a *pivotal base* on either side of the primary sequence.

Figure 4.3 shows the scanning process that MARSs uses to form a potential base pair on the 5' side of the primary sequence. The *open base* on the 5' side of the RNA sequence is used as a pivot nucleotide and *open bases* on the 3' side are sequentially scanned for a potential base pair. The base-pair selection criteria will be explained in the next section.

Figure 4.4 shows a similar scanning process using a *open base* as the pivot nucleotide on the 3' side of the primary sequence and scanning for potential base pairs using *open bases* on the 5' side of the primary sequence.

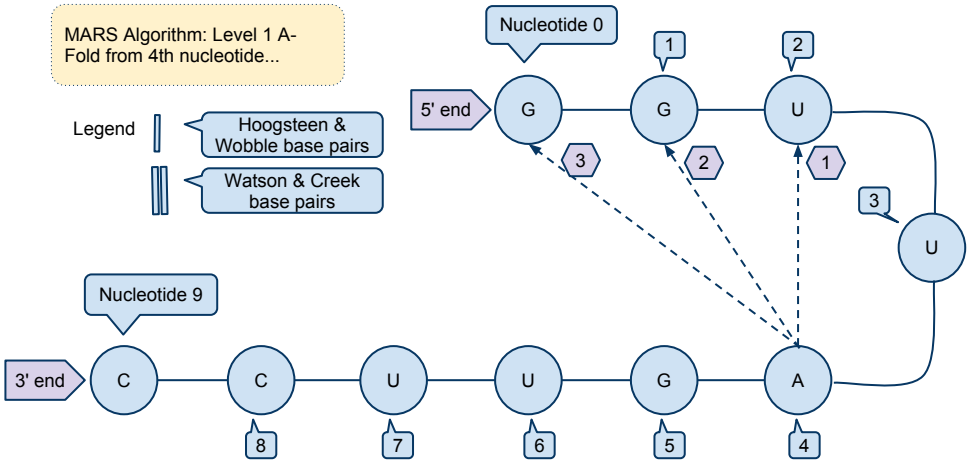


Figure 4.4: MARSs Level 1 Asymmetrical Folding types - 2

4.7 A-Fold Scanning Methods

As briefly mentioned in the previous section, *A-Fold* could occur on either side of the sequence. Therefore, we need to scan on both the 3' and 5' sides of the sequence. Also, a criterion is required for the number of bases to scan and a condition to stop. For this, we introduce two scanning stop methods - first bond and best bond.

First Bond In this option, the scanning process will stop at the first base pair that it can find based on energies for the respective nucleotides in the affinity matrix. The advantage of this option is that the number of free nucleotides that are skipped over (and therefore part of a bulge) will be the least. The disadvantage of this option is that the first base pair may not necessarily be the best (or strong) base pair from all the options that are available.

Best Bond In this option, the scanning process will attempt to make base pairs

with all possible *open bases* on the opposite side of sequence. When a potential base pair is formed, the algorithm will memorize two attributes - the distance of the base pair from the start point and the strength of the base pair and continues scanning. At the end of the scanning, the strongest bonds amongst the various bonds is selected. Should two base pairs of the same strength be short-listed then the one that opens the least number of bases (or closer to the starting point) is chosen. The advantage of this method is that the base pair that is selected is the best among all the possibilities. There are two disadvantages to this method. First, it is computationally expensive as more bases are scanned. Second, it is possible that the bulge due to the free nucleotides could potentially be large when compared with first bond option.

Max Scan The number of *open bases* can be potentially large for a very long sequence. This could have a severe impact on the time complexity of the algorithm. Therefore the algorithm limits the search to a maximum number of bases. The default value is '5' nucleotides as a maximum bulge of 5 nucleotides is observed in Pseudobase [12]. This value can be over written using a configuration option during program startup.

4.8 Base Pair Selection

From the previous two sections, it can be seen that the selection of a base pair is thoroughly explored in order to search for the best possible option. The following bullet list summarizes all the scanning possibilities.

- S-Fold
- A-Fold
  - First Bond - Both 5' and 3' search
  - Best Bond (default) - Both 5' and 3' search

In particular, *A-Fold* scanning is more elaborate and configurable. MARSs can be configured during startup to use *First Bond* instead of *Best Bond*. In addition, scanning along 5' and 3' sides will yield two bonds as well. Therefore, there is a need to select among the short-listed base pairs from these two searches and also between *A-Fold* and *S-Fold* itself.

The selection between the two base pairs predicted from 5' and 3' end searches are analyzed based on the distance from the starting point and strength of the bonds itself.

Distance In this context, distance is defined as the number of free nucleotides between the starting point and the base pair. For asymmetric folds this is easy to count as the nucleotides are along one side of the primary sequence.

It is possible that the two base pairs can have the same distance and for these cases the strength of the bonds are considered.

Strength The strength of the predicted base pair is taken directly from the actual nucleotide interaction values from the affinity matrix. MARSs prioritizes stronger bonds that leave fewer nucleotides i.e., whose distance is as low as possible.

There is also the possibility that both the distance and strength of the short-listed base pair is exactly the same. In this case, one of the base pairs is randomly selected in order to be unbiased about the asymmetry. The selection criteria remains the same whether the *A-Fold* search mechanism is the *First Bond* or the *Best Bond*. There is a disadvantage of randomly selecting the base pair though; two executions of MARSs over the same input sequence might produce different secondary structure predictions. The user can therefore override this randomness by instructing MARSs to choose between either 5' or 3' anchored base pairs. The option is also exposed as a command line argument.

Finally, it is likely that one is a *S-Fold* base pair and the other is a *A-Fold* base pair. Under such a circumstance, the strength of the base pair and the distance is again used for final selection with a minor difference. For distance in *S-Fold* base pair, the number of free bases on one side of the sequence is used instead of both. Through these processes, one final base pair is selected.

The above base pairing process is then repeated by moving one nucleotide on each side of the newly formed base pair and the search is continued for the next base-pair. At the end, the *Level 1* fold is created that could comprise of *S-Folds* and *A-Folds* (on 5' and 3' ends) with free nucleotides interspersed between them. An example structure is shown in Figure 4.2 that comprises of two *S-Folds* between bonds *(0,6),(2,4)*, internal loop between bonds *(1,5)*, hairpin loop at *(3)* and free dangling end from *(7-9)* nucleotides.

4.9 Level 2 Folding

From the previous sections it can be observed that the prediction process is able to predict all of the known secondary structural motifs except pseudoknots and multi-loops without employing a library-based search method. As explained in Chapter 2 pseudoknots can be classified into simple and generic pseudoknots. There are several algorithms that either cannot predict pseudoknots or predict only the simple pseudoknots. This is primarily due to the algorithmic complexity involved in predicting this class of motifs.

MARSs, in contrast, is able to predict pseudoknots and from a design perspective does not distinguish between simple and generic pseudoknots. Therefore, it is able to predict both types of pseudoknots with the same computational complexity. Pseudoknot prediction is the cornerstone feature of the *Level 2* folding while multi-loops can also be produced in certain situations.

Referring back to the *Level 1* structure in Figure 4.2, it can be seen that nucleotides *(7-9)* are both unpaired and are also freely dangling. Free dangling ends are considered problematic in molecular biology as they may bond with other neighboring molecules or simply increase the free energy of the structure being predicted. It is best if free-dangling ends can be avoided.

Level 2 folding is similar to *Level 1* folding and uses *A-Fold* and *S-Fold* techniques to identify secondary structural motifs as well. There are a few key differences though.

The *Level 1* fold may or may not contain a free-dangling end as part of the prediction. Figure 4.2 shows a *Level 1* fold with a free-dangling end and therefore is a good candidate for evolving a secondary structure with a pseudoknot. Figure 4.5 shows the *Level 2* folding containing a pseudoknot using the *Level 1* folding derived in Figure 4.2. Bending the free-dangling end on the 3' side onto the 3' side itself and applying the rules of *A-Fold* and *S-Fold* obtain this structure respectively. One of the notable differences is the selection of the folding point. In the case of *Level 1* folding, the folding point is a set of two free nucleotides. In case of *Level 2* folding, the folding point is the last bonded nucleotide or in other words the entire free-dangling end folds. However, as there can only be single hydrogen bonding per nucleotide, the nucleotide at the folding point cannot bond again. In addition, not all nucleotides are free as well. Many of them would have bonded already as part of *Level 1* folding.

MARSs handles the above situation by simplifying the *Affinity Matrix* and replac-

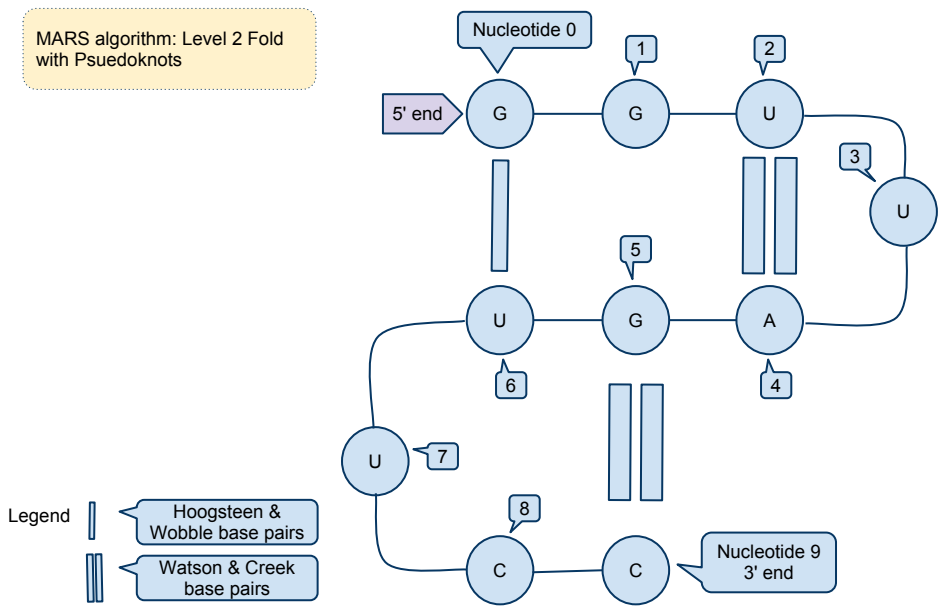


Figure 4.5: MARSs Level 2 Pseudoknot Folds

ing the bonded nucleotides with a value of '0'. The value of '0' was used in *Level 1* folding when bonding was not possible between two nucleotides like A and A. By zeroing the respective cells in the matrix the algorithm marks them as *not possible* as well. The free nucleotides are then arranged facing the bonded side after skipping the minimum number of nucleotides. After this, the search commences using the techniques of *S-Fold* and *A-Fold*.

4.10 Predicting the Final Structures

MARSs can be configured to run in three modes - exhaustive, selective and default. In the exhaustive mode, MARSs will use all of the possible options available such as Symmetric fold, Asymmetric folds (first, best) and two levels of folding.

Naturally, this will result in a lot of predicted structures after two levels of folding. In the selective mode, the user should choose the various configurable options such as folding type to use in both *Level 1* and *Level 2*, the minimum size of the hairpin & internal loops and so on. Consequently, the number of structures predicted is much less but the fidelity of the prediction structures is also directly linked to the selected options. The third and final mode, default, is more like auto-pilot where the algorithm will attempt to choose the best options based on certain properties of the primary sequence. We experimented with sequence length, nucleotides distribution, k-mer histogram values and also if the biophysical determined structure contained a pseudoknot or not.

At the end of the *Level 2* folding stage MARSs has evolved several potential structures for the given RNA primary structure. In *Level 2* stage as in *Level 1* stage there could be duplicate predictions. Therefore, the first step here is to eliminate duplicates. Following this, the secondary structure energies needs to be finalized. The total energy of each of the predicted structure is a summation of the energies of individual base pairs in the predicted structure. Until now, no consideration is given to the effect of the neighboring nucleotides or base pairs. At this stage, the algorithm can be configured to do post-processing and apply any applicable energy models that account for the number of free nucleotides in the hairpin loops, size of stems and so on. By this way, RNA sequences from different species can be processed with different & applicable energy models. Following this, using a threshold energy value, MARSs will filter out the best performing structures. The

threshold values can be specified as an input parameter or the default can be used. The input parameter can be an absolute energy value or a percentile. In the latter case, the top percentile of the predicted structures will be the result and the prediction output. As can be seen MARSs is a highly configurable algorithm and implementation that is meant to be adapted to different RNA worlds. The Figure 4.6 shows the flowchart of the MARSs algorithm.

In Chapter 5 we perform large-scale experiments using parallelized PKNOTS and MARSs on three different HPC hardware architectures. MARSs, being a new and young algorithm, needs to prove that the structures that it predicts are accurate (or close) to the actual known structures. Let us take, for example, a pseudoknotted RNA structure of brome mosaic virus, having a PKB-number of **PKB155** in *Pseudobase* [12]. Figure 4.7 shows one of the high-quality predicted structures for the sequence **PKB155**. All bonds are correctly predicted when compared to the structure in Pseudobase [12], hence the predicted structure has a base-pair distance of 0. It has a PPV of 100% and a sensitivity of 100%. We will define these prediction quality metrics in Section 4.11.

4.11 Prediction Quality Metrics of Interest

We use three accuracy measures to analyze the accuracy of a predicted structure to compare them to experimentally verified structures - **Sensitivity**, **PPV** and **BP Distance**.

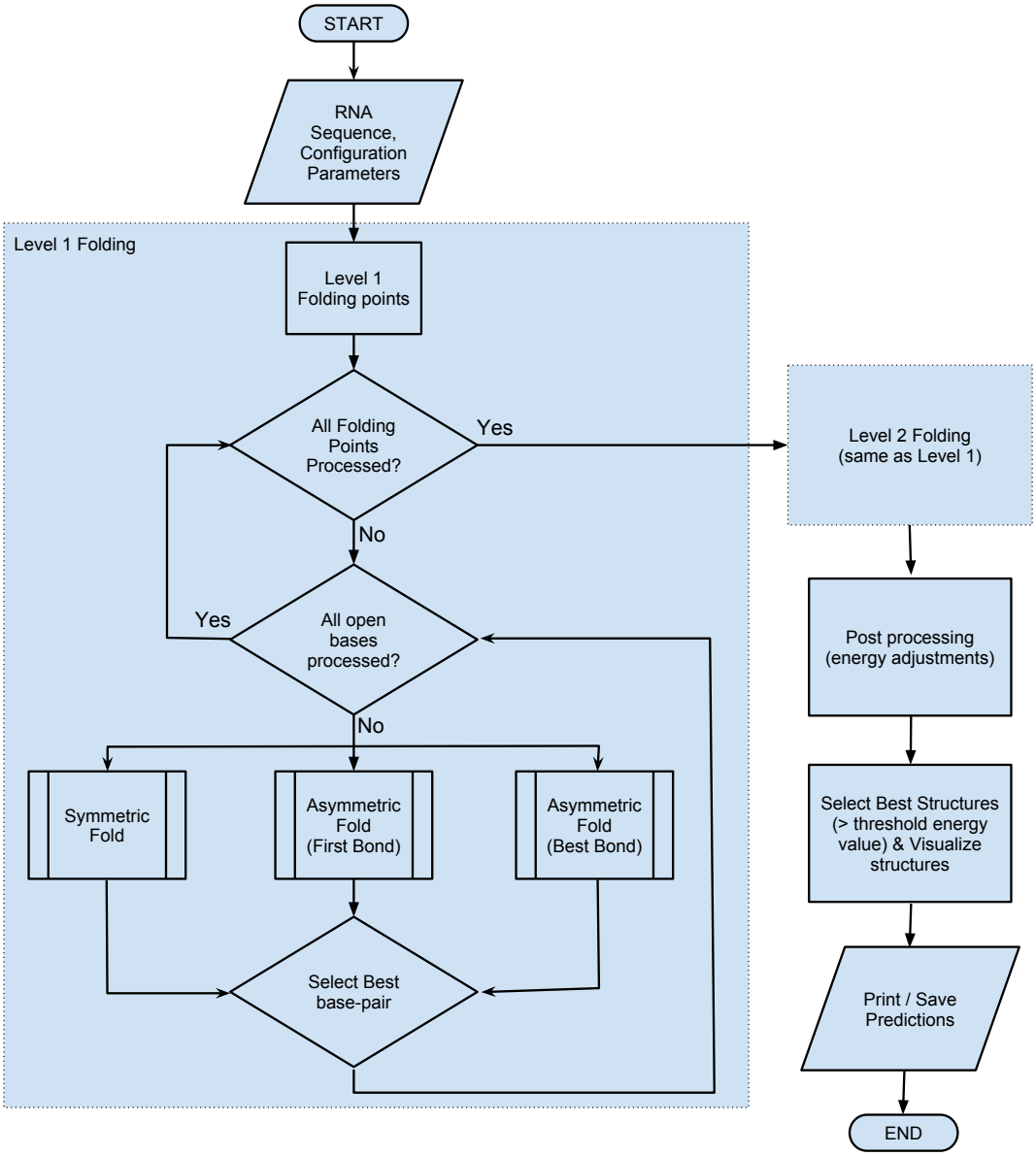


Figure 4.6: MARSs Flowchart

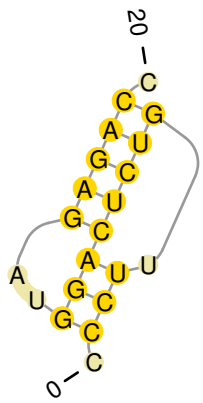


Figure 4.7: One predicted structure of PKB155

Positive Predictive Value (PPV) is the number of correctly predicted base pairs as a percentage of the total number of base pairs in the predicted structure. Its primary focus is on the accuracy of predicted base pairs, without regard to any unpredicted base pairs.

$$\text{PPV} = \frac{\text{number of correctly predicted base pairs}}{\text{total number of base pairs in PREDICTED STRUCTURE}} \times 100\%$$

Sensitivity is the number of correctly predicted base pairs as a percentage of the total number of base pairs in the experimentally verified structure. Its primary focus is on predicting base pairs present in the actual structure, without regards to the number of false base pair predictions. These two measures are regularly used as the standards for measuring accuracy in the case of RNA secondary structure prediction [27].

$$\text{Sensitivity} = \frac{\text{number of correctly predicted base pairs}}{\text{total number of base pairs in ACTUAL STRUCTURE}} \times 100\%$$

BP Distance is the number of different base pairs between the actual structure and the predicted structure. A BP distance of zero means that the algorithm predicted all of the base pairs in the known structure and nothing more.

A structure is perfectly predicted, when both the PPV and sensitivity values are 100%. PPV and sensitivity shows the measure of accurate base pairs predictions relative to the predicted and the actual structure respectively.

4.12 MARSs Complexities

In this section, we derive the time and space complexities of MARSs and also compare the same with some of the relevant algorithms. In Section 4.4 we derived the maximum number of *Level* 1 folds possible and will restate it in Equation (4.1).

$$\text{Maximum number of Level 1 folds} = n(n-1)/2 \tag{4.1}$$

For each of these folding points, the algorithm can be configured to predict base pairs using either Symmetric Fold or Asymmetric Fold (Best Bond or First Bond). Under the exhaustive option, the implementation can be configured to use all these three options and choose the best base pair from among the three outcomes. The maximum and minimum number of potential base pairs in Symmetric Fold can be given by the Equations (4.2) and (4.3). Using these figures, we can arrive

at the average potential base pairs S-Fold transverses and provide it in Equation (4.4). Scanning using the A-Fold technique, under the exhaustive option will simply increase the base pairs to transverse by a constant number, leaving the time complexity the same. Using the folding points and also the number of potential base pairs scanned per folding point we can arrive at the time complexity of MARSs in Equation 4.5.

$$\text{Maximum number of potential base pairs traversed in S-Fold} = \frac{N}{2} \tag{4.2}$$

$$\text{Minimum number of potential base pairs traversed in S-fold} = 1 \tag{4.3}$$

$$\text{Average number of potential base pairs traversed in S-fold} = \frac{N+2}{4} \tag{4.4}$$

$$\textbf{MARSs Time Complexity} = \frac{N(N-1)}{2} \times \frac{N+2}{4} = O(n^3) \tag{4.5}$$

MARSs uses two primary matrices - *Base Pair Matrix* and *Affinity Matrix*. Of these two, *Affinity Matrix* is constructed on the fly and is proportional to the size of the primary sequence. In addition to these two matrices, the software program needs a fixed size of main memory to hold the software instructions and

intermediate data. Therefore, the space complexity of MARSs can simply be given as in equation 4.6.

$$\textbf{MARSs Space Complexity} = O(n^2) \tag{4.6}$$

Although MARSs does *Level 2* folding to predict pseudoknots motifs, the complexity added is either none or not significant. This is because the number of *Level 2* folding points is inversely proportional to the number of *Level 1* folding points and can be explained in two possible scenarios.

Case 1 All the *Level 1* folding points attempted results in base pairs leaving no free nucleotides and therefore *Level 2* folding is not possible. The maximum number of *Level 1* folding points is $\frac{N}{2}$ and since *Level 2* folding is not required the total complexity remains the same. It is also possible that no base pairs are possible in the set of remaining free nucleotides. These type of predictions do not include pseudoknots.

Case 2 In cases where the number of actual *Level 1* base pairs are less than the *Level 1* folding points attempted, the number of *Level 2* folding points is the difference between the number of *Level 1* folding points and the actual number of *Level 1* base pairs realized. This scenario will add to the total time complexity and the value depends on the distribution of the four different nucleotides in the input primary sequence. As the distribution of the various nucleotides differs from sequence to sequence it is difficult to arrive at an

estimate. However, as a general rule of thumb, primary sequences with even distributions of A & U and C & G nucleotides is likely to have more base pairs in *Level 1* resulting in a small runtime for *Level 2* folding.

In general, it should be noted that the actual space and time complexities depends on the composition of the nucleotides themselves that in turn affects the set of possible structures. The resource complexities also depends on the tuning parameters that dictate the final number of secondary structures that are desired. As given in Table 2.1 the algorithm complexity of MARSs is among the best of the algorithms designed so far and is equal to Stormo et al. [16]. It is interesting to note that both MARSs and Stormo's algorithm are non-DP based.

Chapter 5

Performance Evaluation Studies

5.1 Introduction

The focus of this chapter is to describe the various experiments that we have conducted as part of this research study. The experiments were done by parallelizing PKNOTS algorithm and developing MARSs algorithm from scratch. Both the algorithms have been implemented on three different parallel hardware architectures yielding 6 combinations of performance clusters in total. We will rigorously explore the following hardware platforms - Google App Engine, Intel x64, and IBM Cell Broadband Engine. These hardware architectures are described in detail in the Appendices. Appendix A describes the selected cloud-hosted PaaS (Platform as a Service) architecture, the Google App Engine (GAE). Appendix B describes the selected homogeneous ISA & UMA architecture, the Intel x64. Appendix C

describes the selected heterogeneous ISA & NUMA architecture, the IBM Cell Broadband Engine. Each of these appendices provides a generic overview and specific details of the systems such as system architecture, design, limitations and programming challenges. These hardware platforms were specifically chosen as they are fundamentally of different types and in general represent the emerging trends of multi-core processors and cloud computing. In addition, Appendix D recalls some of the HPC architectures that were actively explored by academic researchers and industry alike in the last decade.

This chapter is organized as follows. Section 5.2 describes the master dataset used in our experiments. Section 5.3 describes some of the useful metrics to measure performance gains from parallelization efforts and sets the stage to understand the results from our parallelization experiments. Rest of the sections are coupled into two sections per architecture with each section describing either PKNOTS or MARSs on one hardware architecture. Section 5.4 describes the parallelization efforts and details the performance of parallelized PKNOTS on Google App Engine. Section 5.5 describes the challenges of developing MARSs algorithm on Google App Engine and the performance results. Section 5.6 describes the PKNOTS algorithm's performance on Intel x64 using a physical machine. Section 5.7 measures the performance of running the PKNOTS algorithm on virtualized x64 hardware, as these are typical of an IaaS (Infrastructure As A Service) provider. Section 5.8 describes the performance obtained by MARSs algorithm on the Intel x64 platform. Sections 5.9 describes the parallelization efforts required to port & parallelize

PKNOTS algorithm on the IBM Cell Broadband Engine architecture. Section 5.10 describes the corresponding performance of MARSs algorithm on IBM Cell Broadband Engine architecture. Each combination of algorithm and hardware architecture introduces unique software programming challenges & constraints and also offers distinct performance tuning opportunities as well. Therefore, the runtimes of both the algorithms, especially across the various architectures should not be directly compared value wise. Our objective instead is to understand why the performance of certain hardware and software combination is better compared to others, and in cases where it is low, offer suggestions to improve the same.

5.2 Input Sequence Dataset

We assembled a master RNA sequence dataset for use in our experiments. A total of 1510 RNA sequences were downloaded from publicly available data sources. The data sources were Sprinzl tRNA database [113] assembled by Sprinzl et. al., [83], RCSB Protein Data Bank [114] assembled by Berman et. al., [11], Nucleic Acid Database [115] assembled by Berman et. al., [10] and Gutell lab CRW [116] assembled by Cannone et. al., [17]. The sequence lengths varied from 4 nucleotides to 4381 nucleotides. For all the primary sequences there were either predicted or experimentally verified secondary structures. In addition, the sequences contains both pseudoknots and non-pseudoknots as part of their secondary structures. We used this master dataset across all our three hardware architectures and two

RNA secondary structure prediction algorithms. However, due to constraints such as system availability and speed of data processing we selected a representative subset of this master dataset for experiments on different algorithm & hardware combinations. The representative sequences were selected based on their sequence lengths and also distribution of the various nucleotides.

5.3 Performance Metrics

In this section we derive the performance metrics - Speedup, Incremental Speedup and Performance Gain - as articulated by Gene Amdahl. A standard way to measure performance from a parallelized program is to use the metric of Speedup. Speedup is defined as a ratio of the performance of a given program on a single-processor (or single-core) computer system over the performance obtained in a multi-processor (or multi-core) system. Let the time taken by the original serial program be T_o and the time taken by the parallelized (enhanced) program on the same input be T_e, then the speedup of this software system is given by Equation (5.1).

$$S = \frac{T_o}{T_e} \tag{5.1}$$

Now, we can use speedup to analyze the performance of the paralleled algorithm. Let us suppose that a program P can be fully parallelized and we use n cores to run

this program. Theoretical maximum enhanced performance is given by $T_e = \frac{T_o}{n}$ and the corresponding speedup is given by $S = \frac{T_o}{T_e} = n$. If $n \to \infty$,then $S \to \infty$. However, practically this is unattainable due to parallelization limits and various system overheads.

Let us derive the parallelization gains for a given serial program or algorithm P. For this we will assume that the program is arbitrarily divisible into two parts - one part that cannot be parallelized and has to run serially and another part that can be parallelized. Let us use $|p|$ to denote the number of instructions in P and for simplicity we assume every instruction to take the same number of clock cycles. Let F represent the fraction of the program that can be parallelized. Then the non-parallelizable part is given by $1 - F$. Then the total instructions or total time of the parallelizable part of the algorithm is $|P| \cdot F$. The total instructions or total time of the non-parallelizable part is $|P| \cdot (1 - F)$. Thus the total time taken by the enhanced program can be given by Equation (5.2) where n is the number of cores used to run the program. Now we can calculate the overall speedup of the parallelized program and denote it in Equation (5.3). This equation is a form of Amdahl's law [1] and was originally proposed to measure the performance of a parallel hardware system. We use Amdahl's law to measure the performance of parallelized algorithms on a variety of parallel architectures - homogeneous multi-core, heterogeneous multi-core and auto-scaling cloud platform. Additionally, it is also our interest to quantify the parallelism exposed during runtime, due to the availability of hardware parallelism. Specifically, our interest is to know how much

hardware parallelism that the parallelized parts of one particular algorithm use ? Can the algorithm scale indefinitely should the amount of hardware parallelism is unlimited, such as in auto-scaling cloud platform. We can also derive an upper limit of the speedup and denote it by Equation (5.4). This is a hard limit no matter what parallel computing technology is used.

$$T_e = |P| \cdot (1 - F) + \frac{|P| \cdot F}{n} \tag{5.2}$$

$$S = \frac{|P|}{|P| \cdot (1 - F) + \frac{|P| \cdot F}{n}} = \frac{1}{(1 - F) + \frac{F}{n}}, 0 \leq F < 1 \tag{5.3}$$

$$n \to \infty, S \to \frac{1}{1 - F} \tag{5.4}$$

Using Equation (5.3) we can derive the relationship between speedup and the number of cores used/needed. Figure 5.1 shows a plot on the relationship between 'S' and 'n' at F = 0.1, 0.4, 0.6, 0.8, 0.9 separately.

The patterns in the Figure 5.1 can be split into two distinct regions - a semi-linear and a saturation region. In the semi-linear region, the speedup increases as the number of cores used increases. In the saturation region, using additional number of computing cores in a computer system has little or no effect on the speedup. When F = 0.8 i.e., at 80% parallelization the semi-linear region roughly lies in 0 to 10 cores. After 20 cores, there is virtually no need to add additional computing

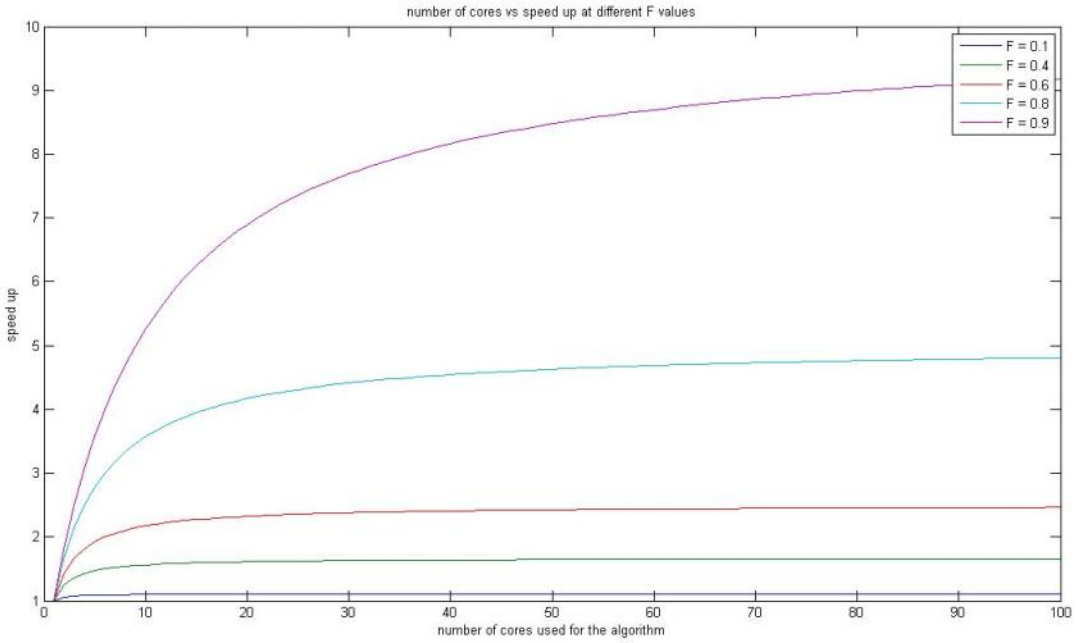


Figure 5.1: Expected Speedup Vs. number of core used at different F values.

cores anymore. Another noticeable trend is that the semi-linear region grows in tandem with the increase in the F values. Therefore, a computer system with many independent computing cores is only useful when you have a very well parallelized algorithm. The secondary structure prediction algorithm and associated software in this study are all highly parallelized and as the number of computing cores are limited in our experimental system, at this stage our hypotheses is that only the semi-linear region will be observed in our experiments.

Equation (5.3) gives us an insight on the relationship between number of computing cores used to run an algorithm and the resulting speedup, which is the performance gain. This equation also indicates diminishing returns when the number of cores increases for a fixed F value. This provides us with guidance on the ideal number of computing cores to run any parallelized algorithm. We will define two more

performance metrics - incremental speedup and performance gain.

Incremental speedup is defined as the difference between successive speedups obtained by adding one more computing core. Performance gain is defined as the ratio of the incremental speedup over the base speedup. Both these metrics are given as Equations in (5.5) and (5.6).

$$\triangle S = S_{n+1} - S_n = \frac{F}{(1-F)^2 n^2 + (1-F^2)n + F} \tag{5.5}$$

$$p_{gain} = \frac{\triangle S}{S_n} = \frac{(F-F^2)n + F^2}{(1-F)^2 n^3 + (1-F^2)n^2 + Fn} \tag{5.6}$$

We use Equation (5.6) to compute the performance gain while increasing the number of parallel computing cores, at various F values. The resulting values are plotted using both linear and semi-log scales in Figures 5.2 and 5.3 respectively. From these graphs we can see the diminishing gains that are obtained as more parallel computing cores are added for a given F values. We can then use heuristics to decide on adding additional computing cores. We use these metrics to understand the results that we obtain from our experiments.

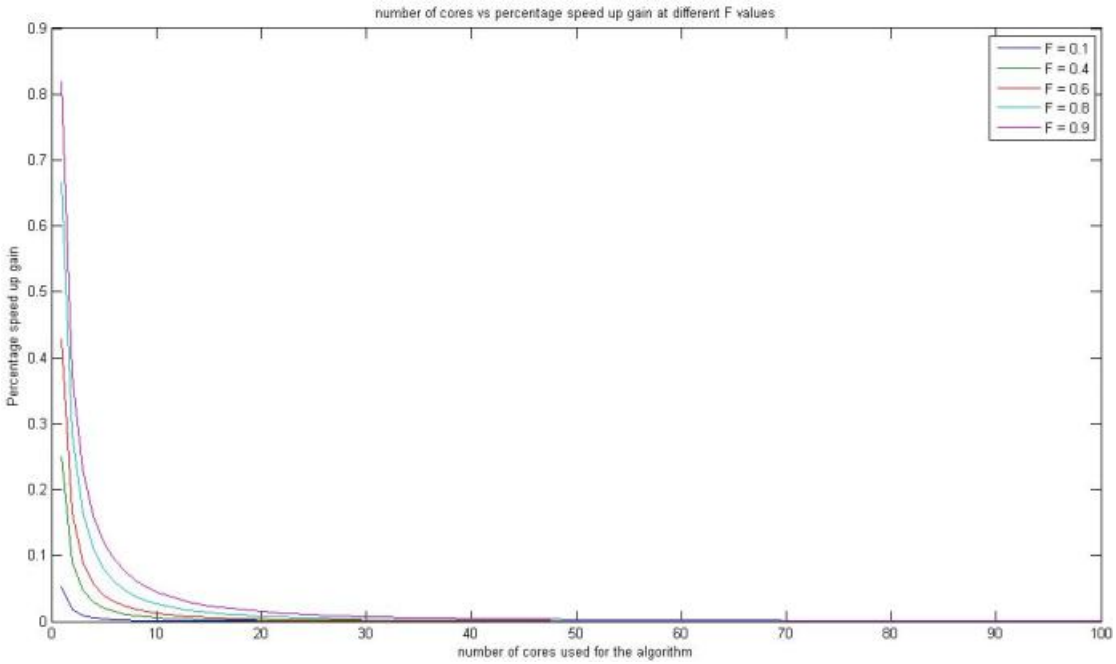


Figure 5.2: Performance gains at different F values

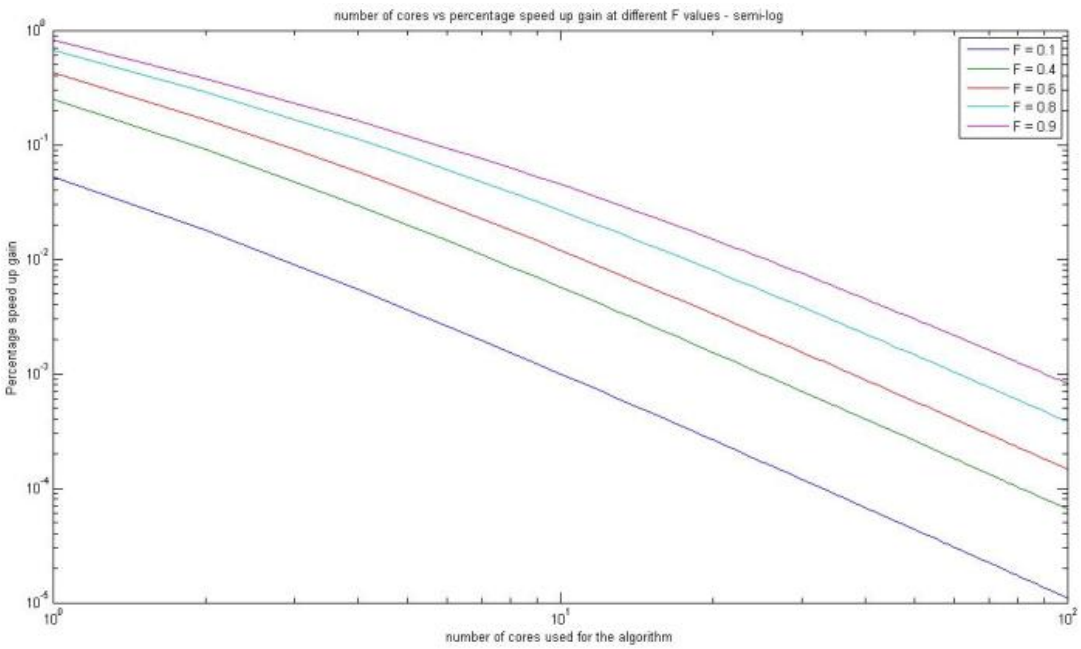


Figure 5.3: Performance gains (using semi-log) at different F values

5.4 PKNOTS on Google App Engine

Google App Engine (GAE) is a new scalable cloud-computing platform from Google. We explore this platform and investigate if it is indeed suitable for handling PKNOTS/MARSs type of HPC algorithms. The platform is currently under "developer preview" and therefore is rapidly changing. In fact, the platform evolved multiple times during the course of our experiments as well. At the time of writing this thesis, the GAE supported two programming language runtimes - JVM and Python. In addition, several programming languages that can be compiled into platform neutral Java byte-code are supported on the JVM as well. We used Python in our experiments and the Python SDK version was 1.4.3.

GAE is a Platform as a Service (PaaS). PaaS architecture abstracts the lower-level infrastructure elements such as networking, computing, storage and exposes an integrated platform to the application. These are offered through standard APIs. In GAE, tasks are the primary means through which parallelism is realized. Tasks are simply functions that are executed by a sandboxed instance (equivalent to a VM process). Each instance is guaranteed a pre-defined amount of system resources (CPU slice, RAM). Other concurrent instances are provided their own system resources, and may be launched in the same physical server (if resources are available) or in a different server. The task scheduling mechanism are unique & different from typical systems and our other architectures used in this study. The tasks cannot store any content on the local system's file system; instead

a network-based managed storage, called datastore, is provided and is available through APIs. Each object stored in the datastore is limited in size and the database type is NoSQL. GAE also provides a volatile in-memory NoSQL style database for storing temporary and data that can be regenerated. The advantage of using memcache is the lower latencies, when compared to non-volatile datastore. Figure 5.4 shows the GAE system architecture and the various resource limitations.

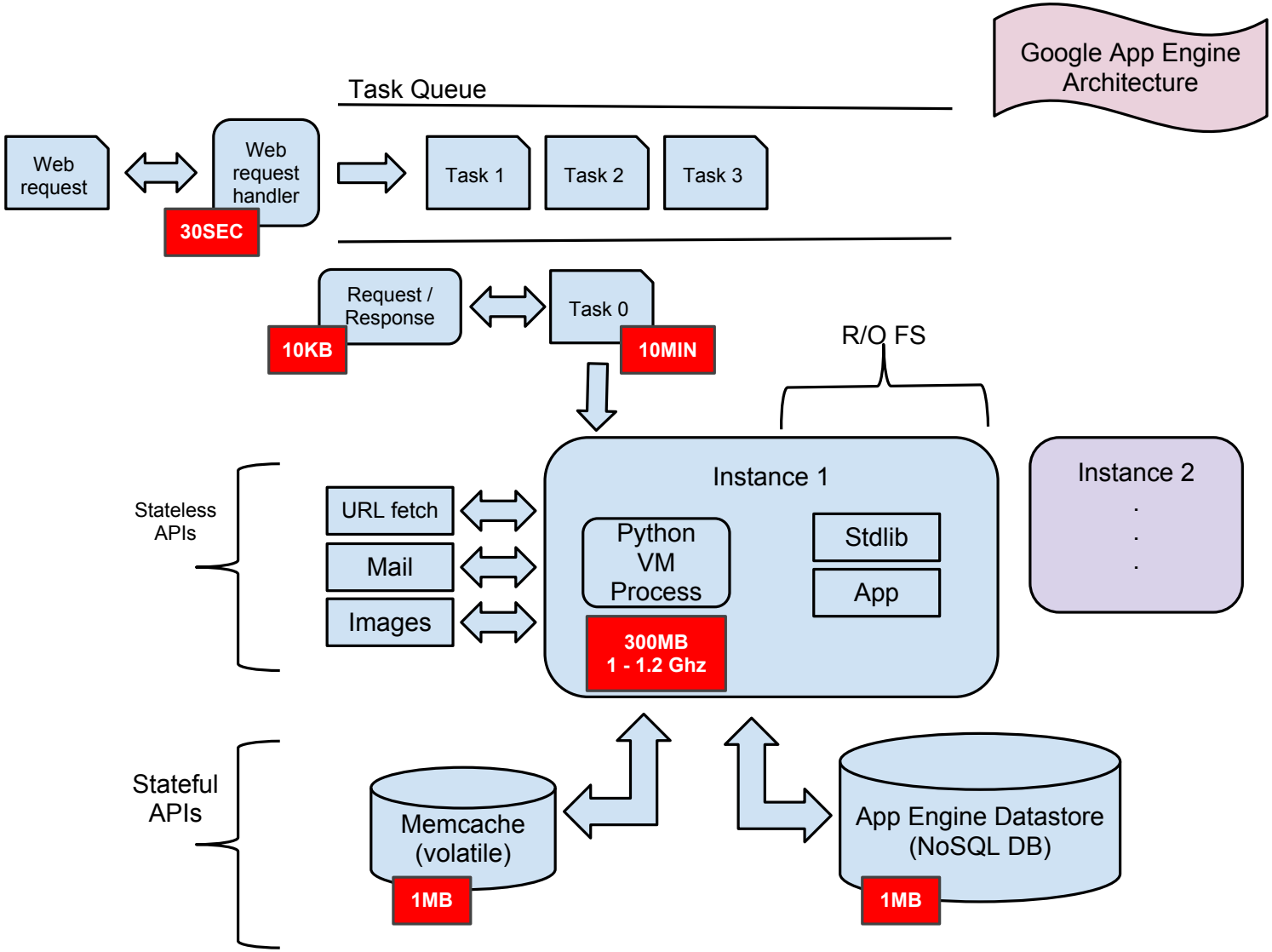


Figure 5.4: Google App Engine - System Architecture & Resource Limits

We next describe our efforts into optimizing PKNOTS to run on GAE and discusses in detail the experimental results. We begin by describing the major challenges - handling space complexities, time complexities - and following this we share the results from executing PKNOTS on GAE.

5.4.1 Challenge 1 - Handling Space Complexity

The first challenge of making PKNOTS available on Google App Engine (GAE) is handling the space complexity. We provide a detailed write-up on GAE in Appendix A. In general, GAE system architecture is very different from a normal Intel based workstation. In particular, programs running on the GAE cannot write to a local file system directly and the data objects themselves cannot be of arbitrary size. The memcache is not designed to be an automatic & transparent caching layer as with CPU caches. The software process address space for both code and data is very limited and finally the latencies incurred to read/write non-perishable data objects varies and are not published. These are the major constraints that needs to be considered in the program design and therefore have a direct impact on algorithm runtimes.

Although the GAE system architecture makes the program design & implementation challenging, it helps in automatically (parallel) scaling the program. The following list highlights some of the benefits of these challenges.

1. By using an API to read/write non-volatile data objects, program no longer

needs to worry about disk access contention and inode exhaustion.

2. The data object size limit of 1MB is to ensure that the latency is low and also that the object in its entirety can fit in both the memcache and also the main memory. Conversely, in an IBM PC as there is no such limit, main memory trashing and cache-miss are commonly known system issues.

3. The optional memcache can be used to selectively store important objects and not cache every object that is read/written from the datastore, which might introduce hidden system performance issues.

4. By having a fixed pre-allotted address space means that each task will not run out-of-memory in the middle of processing.

From the above list, it can be seen that Google has designed the GAE system architecture so that applications can scale and the system constraints actually help to produce better structured & high performing applications.

The total algorithmic space complexity of PKNOTS is polynomial and is equal to $O(N^4)$. The data is split in multiple matrices and the smallest of these is $O(N^2)$ while the largest is $O(N^4)$. As each of the datastore object is limited to 1MB in size the matrices needs to be sliced into multiple objects. In addition, the GAE datastore is a key-value style NoSQL DBMS and this means that the matrix data needs to be stored as a set of key-value pairs. We experimented with several data splicing methods, each with different own pros and cons, and eventually selected two splicing schemes. We call these splicing schemes as simply *'3+1'* and *'1+1'*

and use it for different matrix types as explained below.

'3+1' data scheme We use the *'3+1'* scheme for the matrices that are four-dimensional in nature and therefore $O(N^4)$ in space complexity. In these matrices, the first 3 dimensions are used as the *'key'* for the datastore objects and the data in the 4th dimension is stored as the value as a list object. For instance, for matrix *YHX[i][j][k][l]* the keys will be *YHX[i][j][k]*, with each of the indices varying between '1' and length of input sequence 'n', and the value is a list of numbers in the *'l'* dimension. Let us take an example where the length of the sequence *'n'* is 100. Using this *'3+1'* scheme, there will be 100^3 (or 1,000,000) datastore objects with each datastore object of size 800+ bytes (100 X 8 Bytes per object). Conversely, as each datastore object can be of size 1MB (1,048,576 bytes) we can derive the maximum value of sequence length *'n'* to be 131,072 or simply 128K. For this maximum sequence length of *'n'* there will be correspondingly n^3 objects or $2.25 \times E^{15}$ objects per matrix. Whenever a single matrix cell value is required, the object containing the entire 4th dimension and indexed by the first 3 dimensions is retrieved, value updated and written back to the datastore. As the algorithm computes only one matrix cell at any one time, this method also indirectly conserves main memory usage as the newer objects can replace older objects, making the algorithm scale for longer sequences as well. This access & update method may be seen as a minor drawback, but is currently the best method for maximizing resource-usage and minimizing the communication-overhead

and is further explained in the following paragraphs.

Other schemes In addition to the previous data storage method, we have also considered alternate storage schemes of *'1+3'*, *'2+2'* and *'4+0'* for storing these 4D matrices. As with the *'3+1'* scheme, the first number represents the number of dimensions that are used as datastore object keys and the second number is the dimensions stored as value in these datastore objects. In the *'4+0'* scheme, the datastore object value is simply the 8 byte value from the matrix cell and the number of objects will be n^4. As each datastore object can be up to 1MB in size this scheme will be resource-wasteful and the communication costs in reading/writing an object is likely to be expensive as well. In the *'2+2'* scheme, again using the maximum size of datastore object as a constraint, we can derive maximum sequence length *'n'* to be 362 using the equation $\sqrt{1048576/8}$ and the number of data objects is n^2 or 131,044. Similarly, in *'1+3'* scheme the maximum sequence length *'n'* will be 50 using the equation $\sqrt[3]{1048576/8}$ and the corresponding number of datastore objects will be 50. The longest RNA sequence in our collection is close to 4,500 nucleotides and from the above it can be seen that neither *'1+3'* nor *'2+2'* will be sufficient and *'4+0'* will be resource-wasteful. Therefore, we have chosen *'3+1'* to be the datastore object scheme. Google promises that GAE datastore will be able to scale infinitely. It is important to highlight the fact that this storage scheme will be able to handle all known RNA sequences as none have been identified to be of length 128,000 but some of the discovered

RNA primary sequences are longer than 4,500 nucleotides.

In addition to the 4-dimensional matrices for predicting pseudoknots, the algorithm also needs two 2-dimensional matrices for predicting non-pseudoknots and other supporting 2D matrices as well. For these matrices, we simply use the *'1+1'* scheme in order to balance the resource-usage and communication-overhead. In addition to this data modeling, we have also optimized the storage of ICFG matrix. The ICFG matrix is of size 4276×4276 and contains integer log form grammar for alignment and is used for filling up of the gap matrices. We observed that, although there are 18 million values in this matrix, 99.9% of them are zeros. In this case, we only store the elements that are non-zero and the object key is made of both the dimensions *[i][j]* using the *'2+0'* model. We use operator overloading to check if the object exists in the datastore and if so fetch the object/value, if not we simply return the value '0'. This further reduces the memory requirements.

We have also catered for geospatial access patterns in the matrices, and attempted to further reduce the access latencies in general as well. To achieve this, we have tapped on the optional volatile memcache storage system in GAE. As mentioned earlier, memcache is not an in-line caching layer like a CPU cache. Instead, it has to be activated specifically and different objects can be stored in memcache in addition to the datastore. However, being volatile by design there is no guarantee that objects will be present forever. Therefore, our program stores a copy of the read/written objects in memcache in addition to the datastore. During a data read, the object is first checked in the local storage (AKA main memory), followed

by memcache and if not present in either of these volatile memories it is finally retrieved from the datastore. When the object is not found in both local storage and memcache, the object is retrieved from the datastore to the local storage and a copy is also saved in the memcache. During a data write, the object is stored in the memcache in addition to the datastore. This process will potentially reduce latencies for future data access although results in increased code complexity.

5.4.2 Challenge 2 - Handling Time Complexity

The second challenge in making PKNOTS available on GAE is handling the time complexity. This is primarily because of execution time limits imposed by Google App Engine. GAE is a task-based software platform and by default a computational task can run for 10 minutes only. After the expiration of this time limit, if the task is still incomplete, then it needs to re-spawn itself (by queuing up a new task) and have an internal state-saving mechanism to continue from where it left off. This is usually not a problem for PKNOTS as each of the task that calculates one matrix cell runs from few seconds to few tens of seconds only. As a first step, we have ported the PKNOTS algorithm to GAE but it still runs in serial mode where only one task runs at any given point in time. The whole problem is solved by the single task re-spawning itself again and again at the end of 10-minute boundary. Subsequently, we have experimented with several parallelization methods and sequentially improved the parallelization performance in the process. We call these methods as macro, micro and max parallelization and will be explained

in the next couple of paragraphs.

The process of getting PKNOTS up and running in GAE, required two major steps.

1. To begin with, the entire PKNOTS algorithm was recoded using Python programming language. This is necessary as the reference source code is in ANSI C while GAE only supported Python and JVM (Java Virtual Machine) supported programming languages; ANSI C was not one of them. The PKNOTS-Python version was tested for correctness with the original PKNOTS-C version using the same input sequence on a standard IBM PC running Ubuntu Linux OS.

2. The PKNOTS-Python was subsequently ported to run on GAE taking into consideration the GAE constraints like datastore, memcache, local storage, task-based scheduling and web-based UI. This is still the serialized version, and we call this the 'sequential' implementation because only one task runs at a time and it sequentially computes all the matrices values one after another. The implementation uses a timer to trigger an alarm routine towards the end of allotted time that then saves the current state to both the datastore and the memcache before scheduling the next task. A task's runtime is consumed for both algorithmic computation and also for the synchronous communication to read/write datastore and memcache objects. The implementation differentiates between CPU times spent on algorithmic progression, infras-

tructure overhead activities and keeps track of both of them.

In the sequential version, although PKNOTS is running on GAE it is still executing in serial mode. In addition, as the GAE runs the application in a sandbox, a virtual machine by design, the expected performance therefore is worse than running natively on an IBM PC. In addition, Google states that the CPU clock speed in the hardware powering GAE is between 1.0 and 1.2 GHz. As most of the current desktop workstations are now in the range above 2 GHz, this also means that the performance of any serialized program in GAE is likely to be less than a standard workstation. Following these observations, we explored the parallelization methods available. In any parallelization model, the parallel processes (or threads) need a way to communicate with one another and synchronize their actions. This was our first roadblock as tasks running on GAE cannot communicate with one another, as each one is running in its own sandbox, and the platform doesn't provide any synchronization routines as well. So, we improvised a custom solution for our algorithm.

We used a combination of memcache & datastore to implement a barrier synchronization primitive. In this model, *'i'* parallel tasks are spawned in every computation cycle. At the end of their computation each of them increments a common memcache counter object. In addition, they also write *'i'* different & unique objects to the datastore. By using different objects, datastore contention is avoided. A *'i+1'* th task is started in every computation cycle and checks if the value of the common memcache object has reached *'i'* and if it does then starts the next

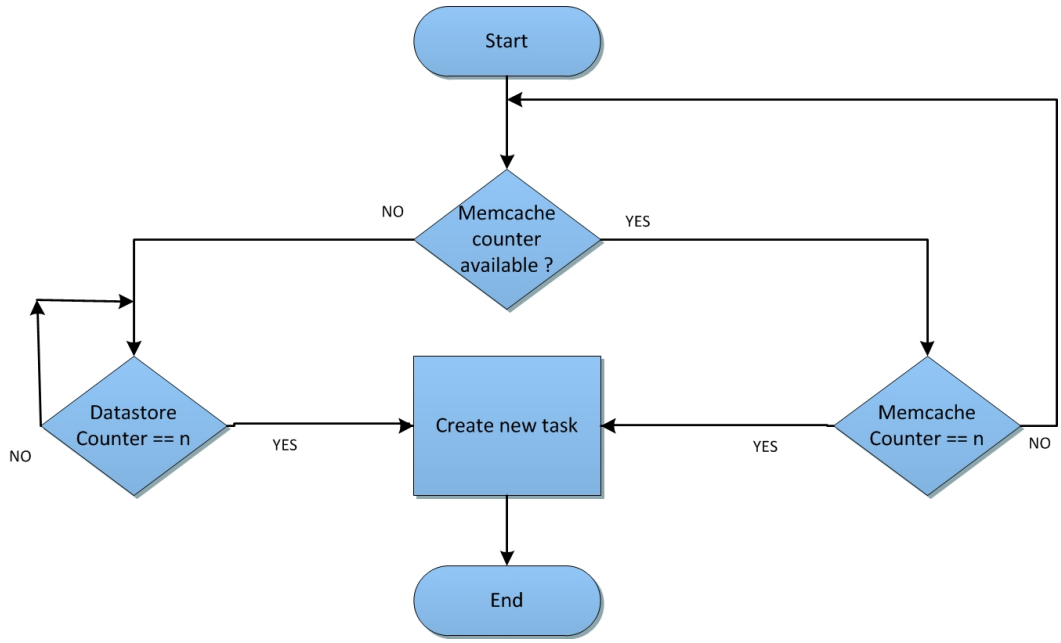


Figure 5.5: Improvised barrier synchronization on GAE

computation cycle. As memcache is unreliable, should the object go missing, the synchronizing task simply resorts to counting the number of datastore objects to ensure all parallel tasks in a computation cycle have finished. We observed empirically the memcache objects to be frequently missing as the length of the sequences increases. Figure 5.5 shows this improvised algorithm as a flow chart.

Macro Parallelization

Our first parallelization method is called *'macro parallelization'*. The parallelization technique that we used in here is the Wavefront parallelization. Wavefront parallelization is a technique that is used to expose hidden parallelization in a dynamic programming algorithm. More specifically, this technique identifies independent sub-problems that can be executed in parallel, in a dynamic programming

T1	T2	T3	T4	T5
T6	T7	T8	T9	T10
T11	T12	T13	T14	T15
T16	T17	T18	T19	T20
T21	T22	T23	T24	T25

Figure 5.6: Sequential filling of a 5x5 matrix in PKNOTS on GAE

algorithm. We have identified four sub-problems or 'macros' that can be executed in parallel. These are the routines that fill in 4 pseudoknot matrices - $fillvhx$, $fillwhx$, $fillyhx$, $fillzhx$ - and each of them have a runtime complexity of $O(N^6)$. Under this model, up to *'i'* tasks can be executed in parallel provided they satisfy a relationship *'r'*. Each task fills up the matrices of a particular index - *[j][d][d1][d2]*. The relationship *'r'* is defined such that the sum of the 3rd and 4th dimensions are equal to a constant *'n'*. The number of tasks that can run in parallel will range from '1' to 'n', where 'n' is equal to the length of the input primary sequence. As an example, we will use a small sequence of 5 nucleotides. The pattern of filling up using both, the sequential process and macro parallelization is shown in Figures 5.6 and 5.7.

In the sequential process, the cells (0,0), (0,1), ... (1,0), (1,1) are computed one by one. Under macro parallelization, in the first computation cycle, index (0,0) is computed, during the second computation cycle, indexes (1,0) and (0,1) are computed and so on. Each computation cycle proceeds only when all the parallel

T1	T2	T3	T4	T5
T2	T3	T4	T5	T6
T3	T4	T5	T6	T7
T4	T5	T6	T7	T8
T5	T6	T7	T8	T9

Figure 5.7: Wavefront parallelized filling of a 5x5 matrix in PKNOTS on GAE

tasks in that cycle completes. This is synchronized using our improvised barrier synchronization primitive. The performance and scalability of our approach is coupled with the size of the input sequenced, i.e., there can be more parallel tasks for a longer sequence compared to a shorter one. The pseudocode for the wavefront parallelization is shown in Figure 5.8. In summary, macro parallelism is built on the foundation of data parallelism as each of the wavefront executes the same code but fills up different parts of the matrices.

Micro Parallelization

Our second parallelization method is called *'micro parallelization'*. In the previous method of macro parallelism, one task in a computation cycle fills up one value in each of the four matrices - vhx, yhx, zhx *and* whx. This task cannot be directly split into 4 concurrent sub-tasks to fill up each matrix elements simultaneously, due to the data dependencies between the various matrices. Table 5.9 shows the data dependencies. From this table, it can be observed that in order to fill

Pseudo code for subroutine FillMtx with macro parallelization

```
for j = 0 to seqlen
        for d = mind to j+1
          FillVP(vx, vp, j, d);
          FillWX (wx, vx, whx, yhx, j, d)
          FillWBX(wbx, vx, whx, yhx, j, d)
           for k = 1 to d+1 cobegin
              for d1 + d2 = k ; d1 >= 0 ; d2 >= 0;
                     FillVHX(whx, vhx, j, d, d1, d2)
                     FillZHX(wbx, vx, whx, vhx, zhx, j, d, d1, d2)
                     FillYHX(wbx, vx, whx, vhx, yhx, j, d, d1, d2)
                     FillWHX(wbx,vx,whx,vhx,zhx,yhx,j,d,d1,d2)
                  end if
              end for
           coend for
        end for
end for
```

Figure 5.8: Psuedocode for subroutine FillMtx with macro parallelization

the matrix cell $yhx[j][d][d1][d2]$ and $zhx[j][d][d1][d2]$, the corresponding matrix element $vhx[j][d][d1][d2]$ needs to be filled first. Similarly, to fill $whx[j][d][d1][d2]$ corresponding matrix elements from other three matrices needs to be filled first. From this access pattern, it can be seen that a 3-step task-based parallelism be introduced in addition to the macro parallelism introduced earlier.

1. One matrix element in vhx is filled.

2. The corresponding matrix elements in matrices yhx and zhx are filled.

3. The corresponding matrix element in matrix whx is filled.

This task-based parallelism is shown graphically in Figure 5.10.

	VHX	**ZHX**	**YHX**	**WHX**
VHX				
ZHX	✓			
YHX	✓			
WHX	✓	✓	✓	

Figure 5.9: Data dependencies among the gap matrices in PKNOTS

	Time T = 1	T = 2	T = 3	T = 4
Task 1	FillVHX(...,d2)	FillVHX(...,d2+1)		
Task 2		FillZHX(...,d2)	FillZHX(...,d2+1)	
Task 3		FillYHX(...,d2)	FillYHX(...,d2+1)	
Task 4			FillWHX(...,d2)	FillWHX(...,d2+1)

Figure 5.10: Task Parallelism in PKNOTS on GAE

In an ideal parallel computer, that has zero or negligible overhead in creating new tasks and accessing discrete & random memory locations, the above scheme will produce good speedup when compared to the serialized version. However, all practical systems incur varying amounts of overheads while creating tasks and accessing memory; GAE is no exception. In GAE, it is very expensive to create and schedule a new task. We have observed the latencies to be in the order of seconds in some cases. PKNOTS-GAE, in one invocation, will create $4 \times n^2$ number of tasks where 'n' is the length of the sequence. At its peak, there could be up to $4n$ concurrently running tasks. Again, using n = 100 as an example, this translates to 40,000 total tasks and 400 concurrently running peak tasks. According to the system documentation, GAE will be able to handle this workload. However, in our case each of these tasks depends on the value created by at least one previous

	Time T = 1	T = 2	T = 3	T = 4
Task 1	FillVHX(...,d2) FillZHX(...,d2) FillYHX(...,d2)	FillVHX(...,d2+1) FillZHX(...,d2+1) FillYHX(...,d2+1)		
Task 2		FillWHX(...,d2)	FillWHX(...,d2+1)	

Figure 5.11: Optimized Task Parallelism in PKNOTS on GAE

task. In GAE, there is nothing equivalent of a shared memory communication that two tasks can use to exchange information. We therefore have resorted to using a combination of memcache and datastore to have a reliable mechanism for information exchange. A total of $4 \times (4n^2)$ number of read/writes to both the datastore and memcache is required for this synchronization. Using n = 100 as the sequence length, this translates to a total of 160,000 calls. This became the second bottleneck and a major performance killer.

We have mitigated this situation to a certain extent using our observation that the computation of WHX takes the longest amount of time and is equal to the sum of computation times for the rest of the matrices. Therefore, we decided to use only two tasks - one computing the $vhx,\ yhx\ and\ zhx$ - while the other computes whx matrix. Again using $n = 100$ as the sequence length, this translates to $2n^2$ total tasks, $2n$ total peak tasks and $2 \times (2n^2)$ datastore and memcache read & writes. Performance is indirectly improved by reducing the overhead costs. The timing diagram in Figure 5.11 shows the optimized version of task scheduling on GAE.

Pseudo code for subroutine FillMtx with macro and micro parallelization

```
for j = 0 to seqlen
        for d = mind to j+1
          FillVP(vx, vp, j, d);
          FillWX (wx, vx, whx, yhx, j, d)
          FillWBX(wbx, vx, whx, yhx, j, d)
           for k = 1 to d+2 cobegin
              for d1 + d2 = k ; d1 >= 0 ; d2 >= 0;
                  if k not equal to d+1 then
                      FillVHX(whx, vhx, j, d, d1, d2)
                       FillZHX(wbx, vx, whx, vhx, zhx, j, d, d1, d2)
                       FillYHX(wbx, vx, whx, vhx, yhx, j, d, d1, d2)
                  end if
                  if d2 not equal to 0 then
                    FillWHX(wbx,vx,whx,vhx,zhx,yhx,j,d,d1,d2-1) //parallel with above function
                  end if
              end for
          coend for
        end for
end for
```

Figure 5.12: Psuedocode for subroutine FillMtx with Max Parallelization

Max Parallelization

Our third parallelization method is called 'max parallelization' and simply an amalgamation of both the macro and micro parallelization introduced above. As macro parallelization is based on data parallelization model and micro parallelization is based on task parallelization model, a hybrid parallelization that includes both of them will produce the best overall performance from parallelized PKNOTS. The code listing in Figure 5.12 shows the pseudocode for the max parallelization model.

5.4.3 Performance Results & Discussions

In this section, we detail and discuss the results from our experiments on GAE. We conducted a set of experiments using all the three versions of the implementation - sequential, macro and micro. In the sequential version, only one task is executed

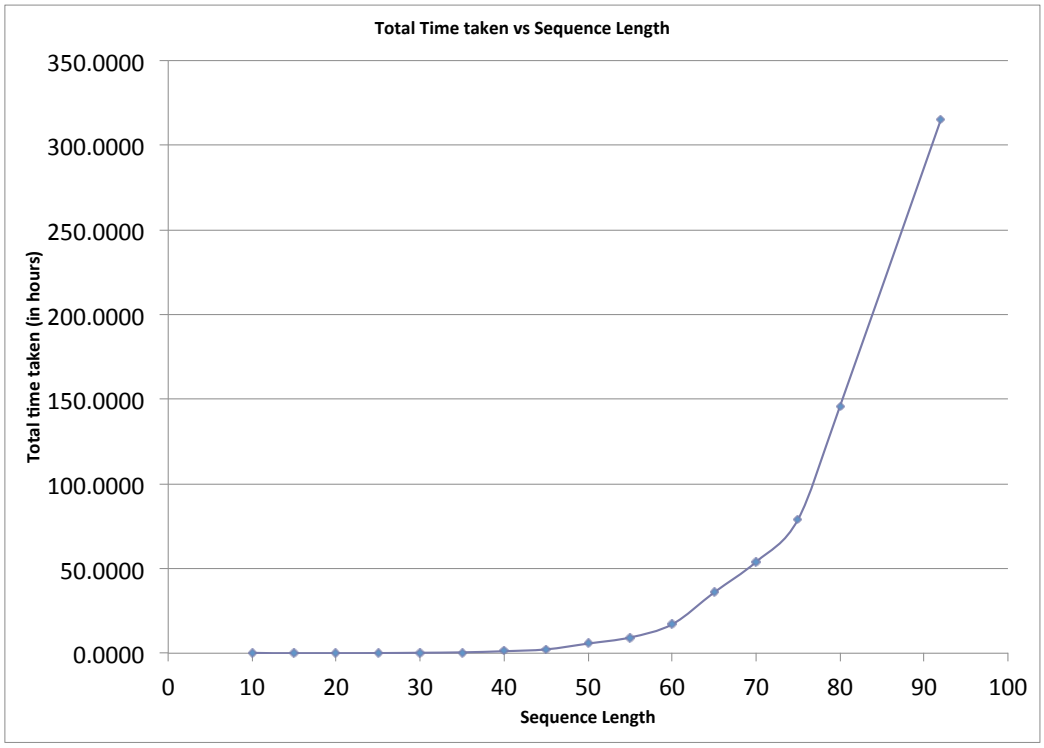


Figure 5.13: Runtimes Vs Sequence length for Serial PKNOTS on GAE

at any one point in time. Each task runs for a full time slice of 10 minutes and then another task is queued that continues from where the previous task left out. In essence, there is no parallelization in this implementation. Figure 5.13 plots the sequence length vs. the run-time. It can be seen that as the length of the sequence increases the run-time also increases, and the growth of the run-time is polynomial as the time complexity of the algorithm is $O(N^6)$. Figure 5.14 shows the same data in log scale where the runtimes for shorter sequences can be seen more clearly and the rate of growth as well.

The total execution time for each instance comprises of algorithmic time and infrastructure overhead time. We define the algorithmic time as the time the implementation spends in calculating and populating the various matrices. Infrastructure

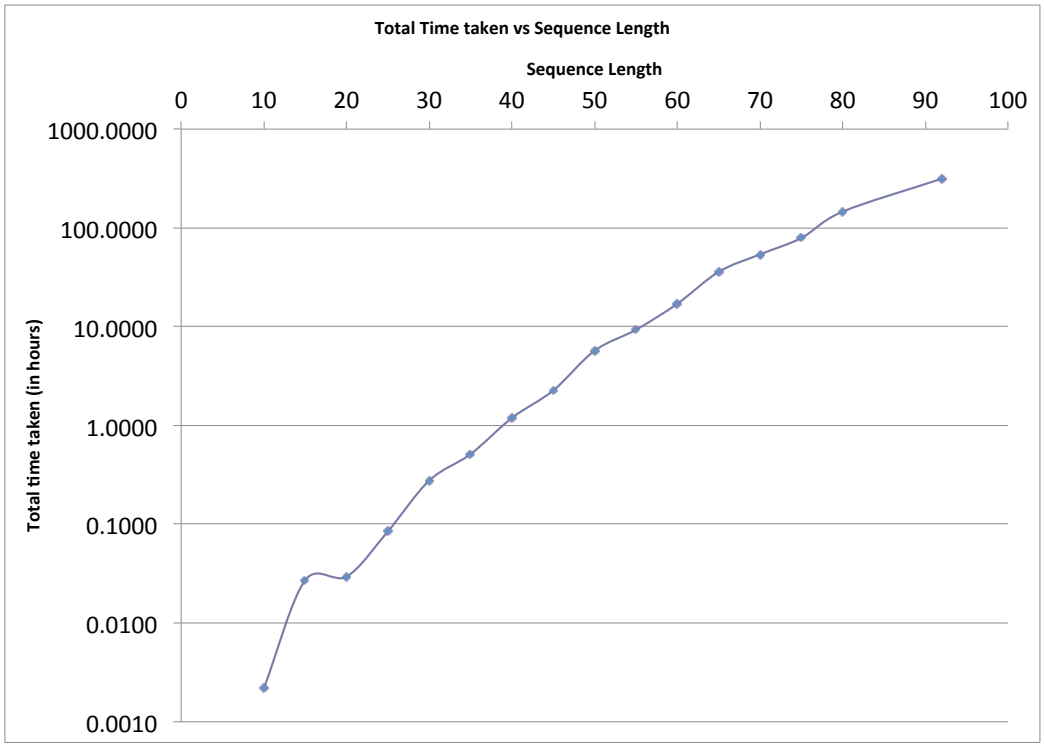


Figure 5.14: Runtimes Vs Sequence length for Serial PKNOTS on GAE - Log scale

overhead time is defined as the time various GAE APIs take to return values. These APIs are used to read/write data to both the memcache and datastore, to create new tasks and for other interactions with the platform. We wanted to find out what percentage of each execution time goes towards algorithmic progression. Using programming language primitives (python in this case), for each of the execution times we measured both the algorithmic and overhead time. As our interest is in algorithmic time, we plot a ratio of algorithmic time to total time vs. the sequence length and show it in Figure 5.15. Several patterns can be observed from this figure.

Sequence length <= 45 As the length of sequence initially increases the ratio of the algorithmic time to total time increases. This is certainly a good

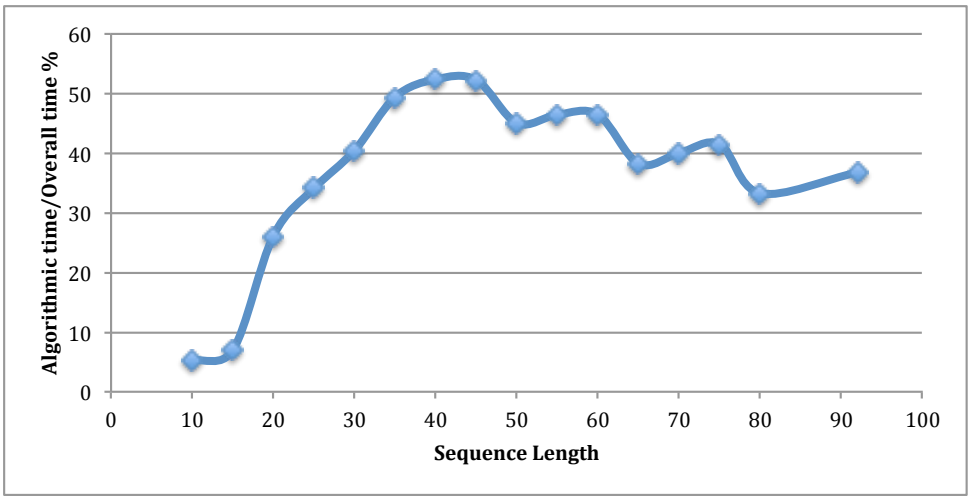


Figure 5.15: Algorithmic Vs Infrastructure Time in Serial PKNOTS on GAE

trait as the program is spending more time towards algorithmic progression rather than on overhead matters. There are a couple of reasons for this. During the early phase of the program execution the data that is retrieved from the datastore is stored on memcache. Subsequently, when new values are computed they are stored in both the memcache and datastore. As the algorithm continues to run, the probability of finding the required value in the memcache increases and therefore the algorithm progresses faster, making the algorithmic portion of the time consumed to be higher. Another important factor is the fixed-duration nature of the task. As each task runs for 10 minutes, during its lifetime it can process multiple matrix values. As there is a data-dependency pattern between various tasks, the later tasks benefit directly by having the results from previous iterations in the main memory itself.

Sequence length > 45 This upward trend continues until around 45 nucleotides

where the ratio is slightly above 50% mark of total time. After this, the ratio of the algorithmic time begins to fall & oscillate as well. We believe this is because of the nature of memcache sub-system. memcache seems to be using some unpublished heuristics to decide on the lifetime of a memory object. In our experiments, we observed that some of the metrics that memcache system might be using are the amount of data one particular application pushes to memcache, last data object accessed time, last data object updated time, number of main memory references to one memcache object within a time period and so on. The overall result is that as the length of the sequence increases, the probability of finding the data in memcache decreases and therefore time is spent to retrieve the object from the datastore, thus adding to the infrastructure overhead time. This trend continued for the rest of the sequences in our test group and we expect it to continue further as well. The inference from this experiment is that unless an application is parallelized it will not gain (and might even lose) performance by running directly on GAE.

Table 5.1 shows the runtimes recorded for the same dataset when executed using the macro-parallelized and max-parallelized versions of the PKNOTS algorithm. It can be observed that max-parallelized version performs better. Figure 5.16 plots the speedup of the algorithmic times of both the versions. Again, it can be seen that the max-parallelized version performs better, although only slightly better, than macro-parallelized version. The mild performance gain is

Table 5.1: Runtimes of Parallelized PKNOTS on GAE

Sequence Length	Time Taken (hours)	
	Macro parallelization	Hybrid of macro and micro parallelization
10	0.0186	0.0094
15	0.1744	0.1481
20	0.4164	0.3483
25	0.6511	0.5744
30	2.8603	2.5242
35	6.1675	6.1706
40	10.0119	9.6683
45	20.6167	20.5333
50	36.3794	35.8019
60	79.0650	75.8386
70	138.0989	131.2339

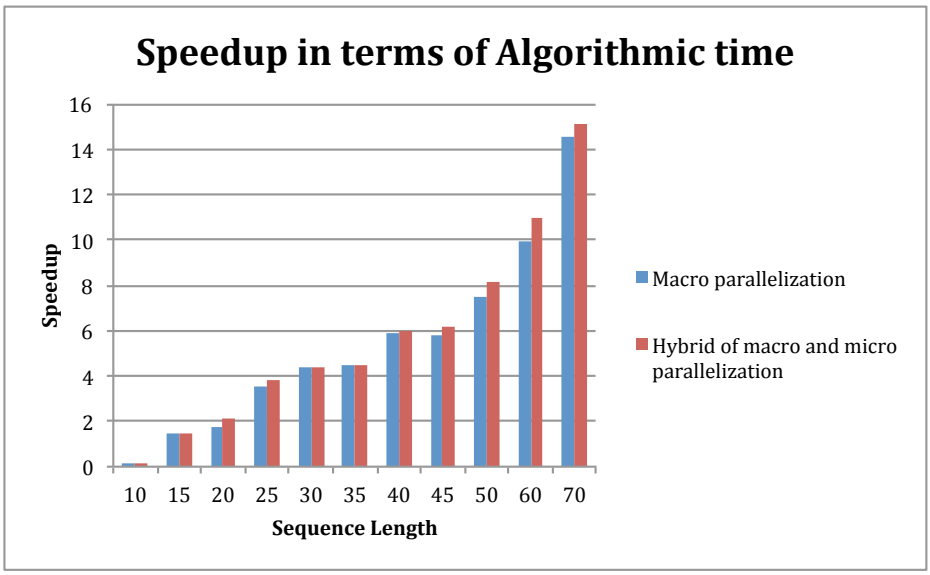


Figure 5.16: Speedup of algorithmic time between macro and max parallelization

because $FillVHX, FillYHX, FillZHX$ occupies only 3% of total runtime while $FillWHX$ occupies 89%. The overall performance gain is attributed to the fact that in max-parallelized version there are more concurrently running tasks compared to macro-parallelized version. As the number of concurrently running tasks is tied to sequence length, with longer sequences better speedup is expected and is observed to be more when compared to shorter sequences. One notable artifact with the execution times of the parallelized versions is that it is slower than the sequential times for the same length sequences. The primary reason for this unexpected outcome in GAE is the notable absence of shared memory and synchronization routines. More specifically, in the parallelized version the lifetime of any task, specifically for shorter sequences, is in the range of few seconds to few tens of seconds. The implication of this behavior is that the local variables are flushed when a new task is created. The new task needs to fetch the values from the datastore or the memcache again. At the time of these experiments, GAE did not have any mechanism to re-use the same address space for running several sequential tasks. This coupled with the fact that for longer sequences the data is also not reliably found in the memcache, degrades the performance even more. We believe this to be a transient situation as Google is rapidly adding newer features to GAE. As an example, when we initially started this project the max runtime for all the tasks was only 30 seconds and was subsequently increased to 10 minutes. Had the max time remained at 30 seconds, the performance gain for sequential version would not be so high after all.

PKNOTS Algorithm for RNA Secondary structure prediction

Sequential Implementation

1. INPUT RNA SEQUENCE :

Length : 0

(OR) CHOOSE FROM EXPERIMENTALLY VERIFIED SAMPLES : None

Non-Pseudoknot Samples :	**Source & *Description***
GGACCGGAAGGUCC (14)	RCSB Protein Data Bank 1WTS *16S ribosomal RNA from B.stearothermophilus*
CCCCGGGGCCCCGGGG (16)	Nucleic Acid Database adh012 *A-Form Variant with an Extended Backbone Conformation*
UCAGACUUUUAAUCUGA (17)	RCSB Protein Data Bank 1BZ3 *TRNA polyribonucleotide*
GGGGAUUGAAAAUCCCC (17)	RCSB Protein Data Bank 1J4Y *Anticodon Stem-loop from E. coli tRNA(Phe)*
UAGCCCCGGGGCUAUAGCCCCGGGGCUA (28)	RCSB Protein Data Bank 435D *E. coli tRNA(Ala)*
GGCUCGUGUAGCUCAUUAGCUCCGAGCC (28)	RCSB Protein Data Bank 1ZBN *BIV mRNA polyribonucleotide*
Pseudoknot Samples :	
CGCGCGCGCGCG (12)	Nucleic Acid Database zdfs33
AGUGGCGCCGACCACUUAAAAACACCGG (28)	RCSB Protein Data Bank 1YG3 *Sugarcane yellow leaf virus (ScYLV) RNA pseudoknot*
UCCGGUCGACUCCGGAGAAACAAAGUCA (28)	RCSB Protein Data Bank 1KPY *Luteoviral P1-P2 frameshifting mRNA pseudoknot*
GGCGCAGUGGGCUAGCGCCACUCAAAAGCCCG (32)	RCSB Protein Data Bank 1KAJ *Non-frameshifting RNA pseudoknot from mouse mammary tumor virus*

2. YOUR REFERENCE ID (Maximum length = 20) :

3. EMAIL ADDRESS :

Submit Reset

Figure 5.17: Screenshot of the serial version of PKNOTS on GAE

We have implemented all three versions - sequential, macro and max parallelization - on the free version of Google App Engine at the URLS `http://pknots1.appspot.com` (sequential), `http://pknots2.appspot.com` (macro) and `http://pknots3.appspot.com` (max parallelization). The availability of these URLs is subject to the non-exhaustion of daily free quotas for GAE web applications. We are also providing a screenshot of the sequential version as a reference in Figure 5.17.

5.4.4 Is GAE an ideal platform for PKNOTS?

Based on our experience of implementing PKNOTS on GAE, the feasible experiments we conducted and the results that we obtained, we arrive at the conclusion that GAE in its current incarnation is not a good fit for PKNOTS type of HPC applications. The following list of shortcomings will justify our observations & claims.

Expensive Tasks The computation cost of creating a new task is very expensive in GAE. This is primarily due to the fact that a new sandboxed VM needs to be allotted, the application byte code should be loaded before execution can commence. In cases where an instance is already running, it needs to be purged of data from previous task execution before a new task can execute.

Synchronization At the time of these experiments GAE lacked any system primitives that provides synchronization functionality like barrier synchronization. The absence of this important system primitive, prevents cooperative tasks from updating other peer tasks of their progress.

Memcache The memcache system is non-volatile by design. In addition, Google does not declare the size per application. These combined together for a HPC application, that tends to write a lot of temporary data, results in high memcache flushing and more calls to the datastore ultimately.

Main Memory The size of main memory per process was also limited to 300MB at the time of our experiments. This resulted in application managing the

data by writing excessive items to the datastore. This process incurred overhead time through the use of CPU time and also latencies to read/write to the datastore.

CPU Speed The CPU speed was limited to between 1 and 1.2 GHz and this resulted in the application running much slower compared to today's standard workstation

GAE++ We observed these shortcomings while using the Python runtime with SDK version 1.4.3. Google has since announced the addition of more HPC application friendly features. Therefore, we believed it is only a matter-of-time before these shortcomings are addressed. We describe some of these newer features in Chapter 6.

5.5 MARSs on Google App Engine

In this section, we discuss our efforts to make MARSs available on the scalable cloud-computing platform - Google App Engine (GAE). MARSs, unlike PKNOTS, does not use deep recursions and therefore is considered to be easier to parallelize. Using our experience with making PKNOTS available on GAE we next make available MARSs on GAE. Following the implementation, we describe and discuss the results that we obtained as well.

5.5.1 Optimizing MARSs for GAE

Unlike PKNOTS, it was much easier to develop MARSs to execute on GAE. This is primarily because MARSs has been designed with parallelization as one of its core-enabling features. Every stage of the MARSs algorithm is designed so that it can be parallelized and with ease on different parallel architectures. On GAE, tasks and task queues is the primary method to introduce parallelism. As such, we use both code and data parallelism in an intrinsic way and at different stages of the algorithm. As described in detail in Chapter 4, MARSs mimics RNA secondary structure formation using two stages - *Stage 1* and *Stage 2*.

Stage 1 In this stage, the algorithm identifies a folding point and enumerates base pairs in a zipping fashion. Several folding points are proposed & analyzed and up to 3 different base-pairing methods can be used. Therefore, there is a clear need for parallelization.

Stage 2 In this stage, a second folding point (if possible) is identified to create a pseudoknot with additional base pairing.

As briefly described above, base-pairing in each stage can be attempted using three distinct processes called - *Symmetric Folding*, *Asymmetric Folding - Best Bond* and *Asymmetric Folding - First Bond*. A lightweight synchronization method is required at two stages of the algorithm - at the base-pair choice stage and between *Stages 1* and *Stages 2*. Performance of MARSs or any other RNA structure prediction algorithm is largely dependent on the distribution of the nucleotides in a

given sequence. A sequence where the four nucleotides are evenly distributed, is likely to have large number of canonical base pairs. At the same time, another sequence of the same length with an uneven distribution of nucleotides is likely to have lesser number of base pairs. Therefore, the nucleotide distribution of a sequence is also indirectly proportional to the run-time of the MARSs algorithm.

MARSs Initialization

An application on GAE begins execution in response to a web request. In the case of MARSs application, the web request consists of the primary sequence of the RNA for which secondary structure(s) needs to be predicted along with any configuration parameters. A handler task, for web requests, is activated and has far more constraints than a regular background task. Chief among the constraints is that it cannot execute more than 30 seconds. Therefore, in our application design the web-request handler simply creates a 'root' task based on the web-content (sequence, configuration parameters) received. The 'root' task is responsible for kick starting the algorithm by creating the *Base Pair Matrix*, creating the *Affinity Matrix* using *Base Pair Matrix* and the input sequence and finally identifying the various folding points in the MARSs algorithm. However, no folding has occurred yet and all the above is done in a serial fashion as parallelism will add very little gain at this stage. At this stage, any suitable thermodynamic model can be used and affinity values between various positional base pairs can be updated in the *Affinity Matrix*. The *Affinity Matrix* is then stored in the datastore as a

single object and will also be cached in the memcache on first access. As each datastore object is limited to 1MB in size, catering to a base-pair affinity value in 64 bits, this model enables MARSs to handle input sequences of up to 362 base pairs in length. We specifically choose this model for two reasons. First, given our experience with PKNOTS and GAE datastore we knew that read/writing from the datastore/memcache is the number one performance killer and wanted to explore another method. Second, MARSs references matrix values more than PKNOTS and at every base-pair stage. Therefore having the matrix as a single object, although with size limits, will help us to extract the best performance possible from GAE.

Kick starting Level 1 Tasks

Next, for each of the folding points identified, the 'root' task creates and queues an independent task to process that folding point. We choose to pass the various parameters as payloads to the newly created tasks instead of making the task fetch from the datastore. As the payload size is limited to 10KB the parameters are the name of the affinity matrix in memcache along with the folding points to be processed. The *Level 1* tasks then fetches the *Affinity Matrix* from the memcache. If the *Affinity Matrix* is evicted from the memcache for any reasons, then the first task that references it will read from the datastore to its main memory and also replicates it in the memcache. After this all other tasks can simply read from the memcache. As it is possible that more than one task can detect, at almost

the same time, that the *Affinity Matrix* is missing from memcache our design uses semaphore-type variable in memcache to notify other tasks that the current task is fetching the *Affinity Matrix* from the datastore. Other affected tasks will simply wait for a fixed number of seconds before retrying. It is to be noted that each task copies the *Affinity Matrix* from the memcache to its main memory before processing. By this way, the number of references to either memcache/datastore is greatly reduced. Each of the *Level 1* tasks is based of the same codebase and processes different data; this design therefore exhibits data parallelism.

Processing Level 1 Folds

Each of the independently executing *Level 1* tasks enumerates the base pairs for a given folding point. Each of the base pairs can be decided using the outcome of three different search routines - *Symmetric Bond*, *Asymmetric − First Bond* and *Asymmetric − best bond*. Each of these sub-routines is different from each other, thereby exhibits code parallelism. We have used this code parallelism approach in our other implementations on Intel x64 and IBM Cell architectures. However, given our experience with PKNOTS on GAE, we decided not to enable it for MARSs on GAE. In order to understand this decision from a technical standpoint, let us derive the number of memcache/datastore reads/writes that will be required. For each of the base pairs that is being enumerated, three tasks must be started and each of these tasks need to read the affinity matrix for processing. In addition, each of the tasks needs to know the state of the base-pair enumeration

as well and this can be passed as a parameter, subject to a cap on payload size. After each of the task finishes, they will have to write the results to the datastore (as memcache is not reliable and a data loss could mean the task has to be rerun) and a watchdog task to monitor the progress of these new tasks and restart the *Level 1* task again. In total, 5 tasks have to be created for every base-pair enumeration, 3 memcache reads and 3 datastore writes performed, unknown number of datastore reads by the watchdog. This whole process needs to be repeated for up to n/2 base-pair enumeration per tasks and for up to n(n-1)/2 fold points that are being checked by individual tasks. As can be seen this is a very complex and performance-killing system design and moreover the gains from parallelizing the individuals sub-routines does not justify the significant amount of overheads.

Kick starting Level 2 Tasks

Each of the *Level 1* tasks complete the enumeration of the base-pair list and write the result to the datastore. In addition to the 'i' *Level 1* tasks that processes the folding points, the root task also creates an *'i+1'* th watchdog monitor task. The monitor task is passed an argument containing the value of 'i' and also the prefix the *Level 1* tasks will use to create the datastore result objects. The monitor task using GQL (Google Query Language) periodically queries to count the number of objects in the datastore that matches the prefix. If there are 'i' objects then all 'i' *Level 1* tasks have completed and the monitor task starts the *Level 2* 'root' task. *Level 2* root task has a set of objectives. First, it analyses the *Level 1* structures

and eliminates (any) duplicates. Let us refer to this reduced number as 'k' where *'k = i - j'* and 'j' is the number of duplicates and non-pseudoknots. Second, for the 'k' *Level 1* structures the affinity matrix is modified using the base pairs to produce 'k' versions of the *Affinity Matrices*. These 'k' *Affinity Matrices* are then used as the new *Affinity Matrices* for determining the *Level 2* folding. This *Level 2* 'root' task then computes the folding points for each of these 'k' *New Affinity Matrices*. Finally, the *Level 2* root task spawns new tasks for each of the folding points in each of the 'k' *Affinity Matrices*. It also spawns one *Level 2* watchdog monitor task. The *Level 2* watchdog will determine if all the *Level 2* tasks are complete and after that will eliminate the duplicates, sort the folding according to the resultant energy and as a final step visualize the RNA secondary structure. Figure 5.18 shows the complete MARSs/GAE processing pipeline.

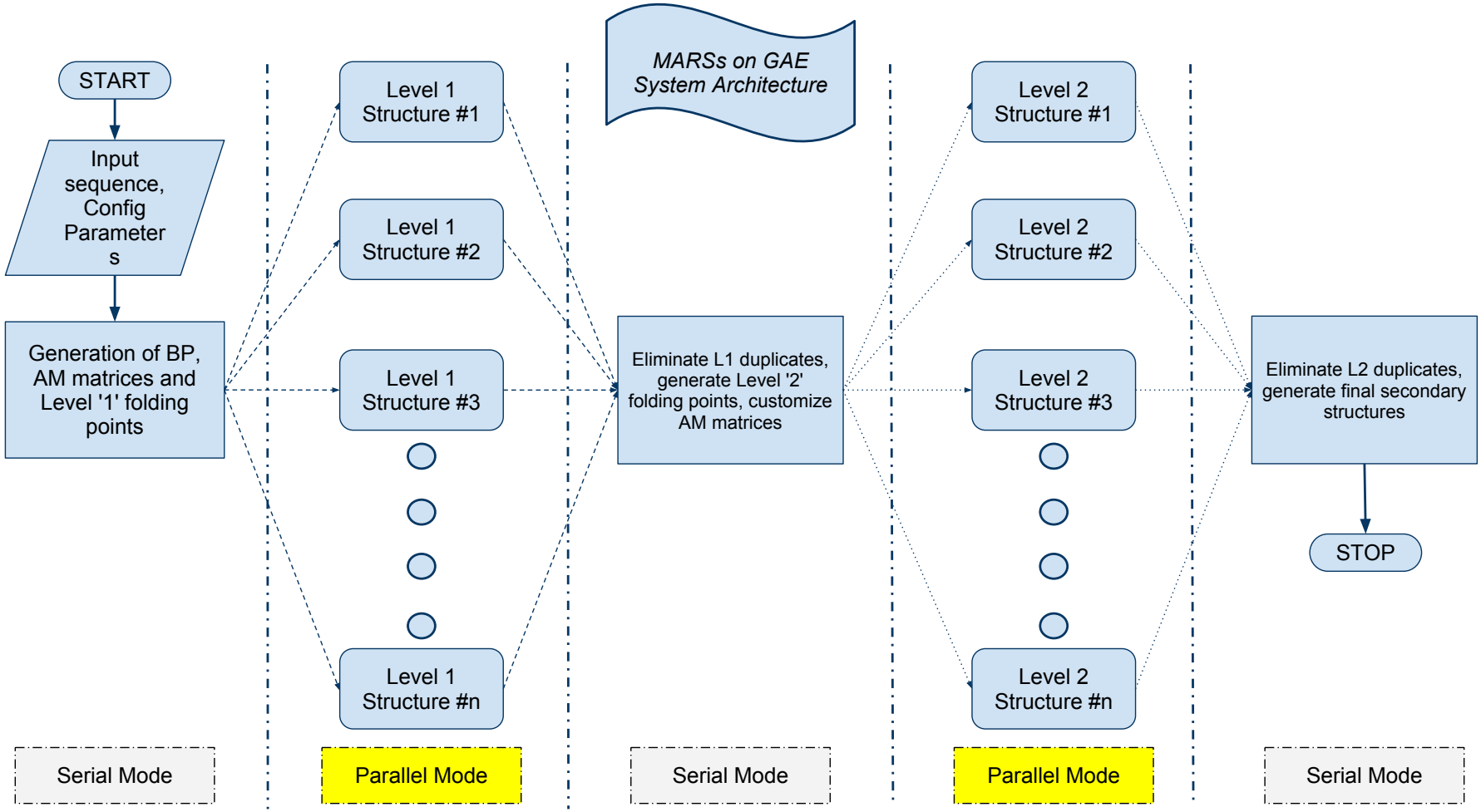


Figure 5.18: MARSs on GAE - Work Flow

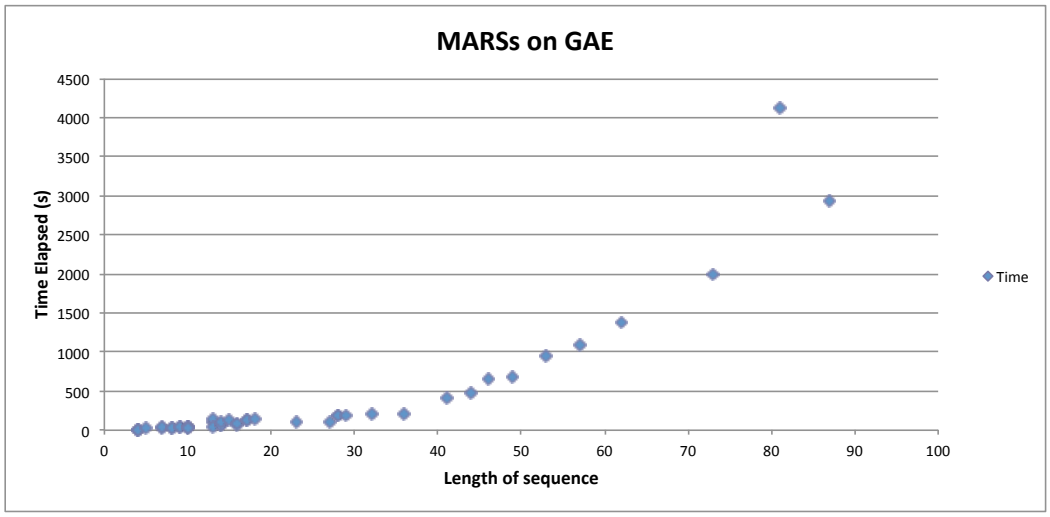


Figure 5.19: Runtimes of MARSs on GAE

5.5.2 Performance Results & Discussions

In this section, we first show and discuss the performance results obtained from MARSs on GAE. This is shown in Figure 5.19. From the figure it can be seen that MARSs performance is an order of magnitude better than PKNOTS. This performance gain is directly attributed to the non-recursive algorithmic structure of MARSs vs. the highly-recursive algorithmic structure of PKNOTS. A non-recursive algorithm is generally expected to perform better in a parallel architecture. At the same time, it also should be noted that most of the performance-loss in PKNOTS was due to the read/write to the datastore while we avoided the same in MARSs. Also, PKNOTS implementation was using different thermodynamic model compared to MARSs. Figure 5.20 plots the runtimes of PKNOTS and MARSs in the same graph using log scale. This shows the performance difference between both the algorithms. It can also be seen that for very small sequence lengths, PKNOTS algorithm is actually faster compared to MARSs.

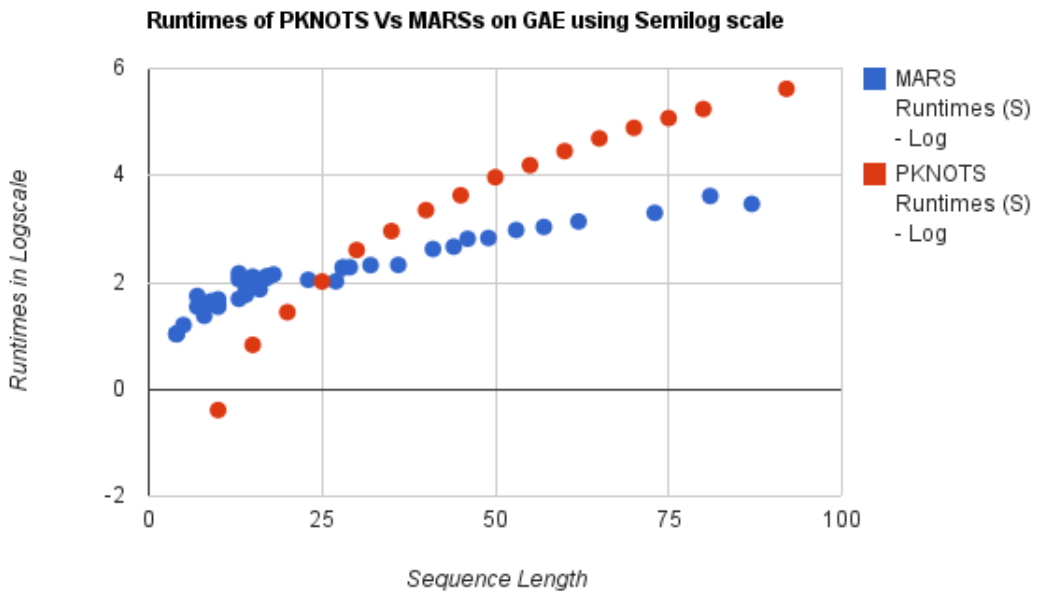


Figure 5.20: Runtimes of MARSs and PKNOTS on GAE

In MARSs, we introduce 3 different base-pair selection methods - *Symmetric*, *Asymmetric Best Bond* and *Asymmetric First Bond*. Although *Symmetric* and *Asymmetric* methods are required to enumerate all the different types of secondary structural motifs, it was not clear to us if the 'best bond' or 'first bond' was producing the structures that are of higher-quality i.e., lower energy. For this, we added in a task trace to each of the *Level 1* and *Level 2* tasks. When *Level 1* tasks produces the various structures, information about the methods used is saved. *Level 2* root task reads this and passes it to *Level 2* tasks, which then add more information to it. At the end of this process, we categorized the various secondary structures & counted their numbers in the top percentile. We found that 'Asymmetric best bonding' produces majority of the top percentile structures. This pattern emerged both in *Level 1* folding and also *Level 2* folding. We show the statistics in Figures

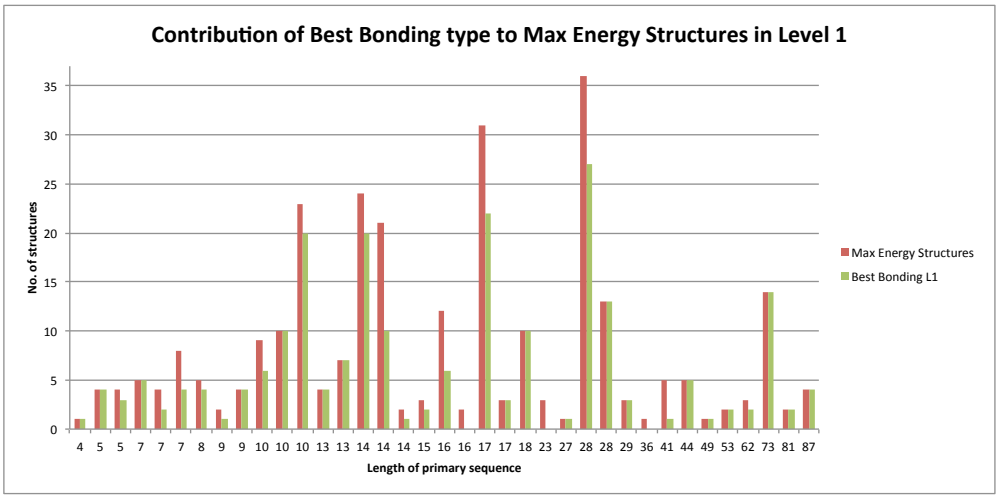


Figure 5.21: Number of Predicted Structures in Level 1 using Asynchronous Best Bond

5.21 and 5.22. It should be noted that with a different thermodynamic model the outcome could be different. We analyzed the performance of MARS from quality perspectives, using a selected set of experimentally verified sequences. The results are provided in Section 5.8. As the results from the experiment are the same on the GAE, we choose not to include them here as well.

5.6 PKNOTS on Intel x64

In this section we describe the results from our experiments with parallelized PKNOTS on the Intel x64 architecture. Parallelization is similar to what was done on the Google App Engine (GAE) platform and therefore to avoid duplication, we opt not to repeat it again. The major difference being that codebase here is in ANSI 'C' while it was in Python on GAE. This difference of using a compiled language (ANSI C) vs. an interpreted language (Python) manifests it-

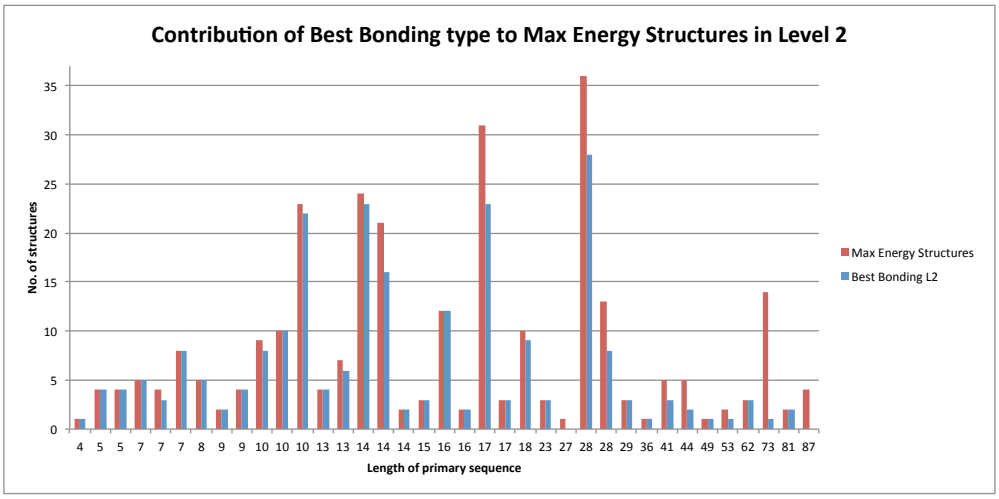


Figure 5.22: Number of Predicted Structures in Level 2 using Asynchronous Best Bond

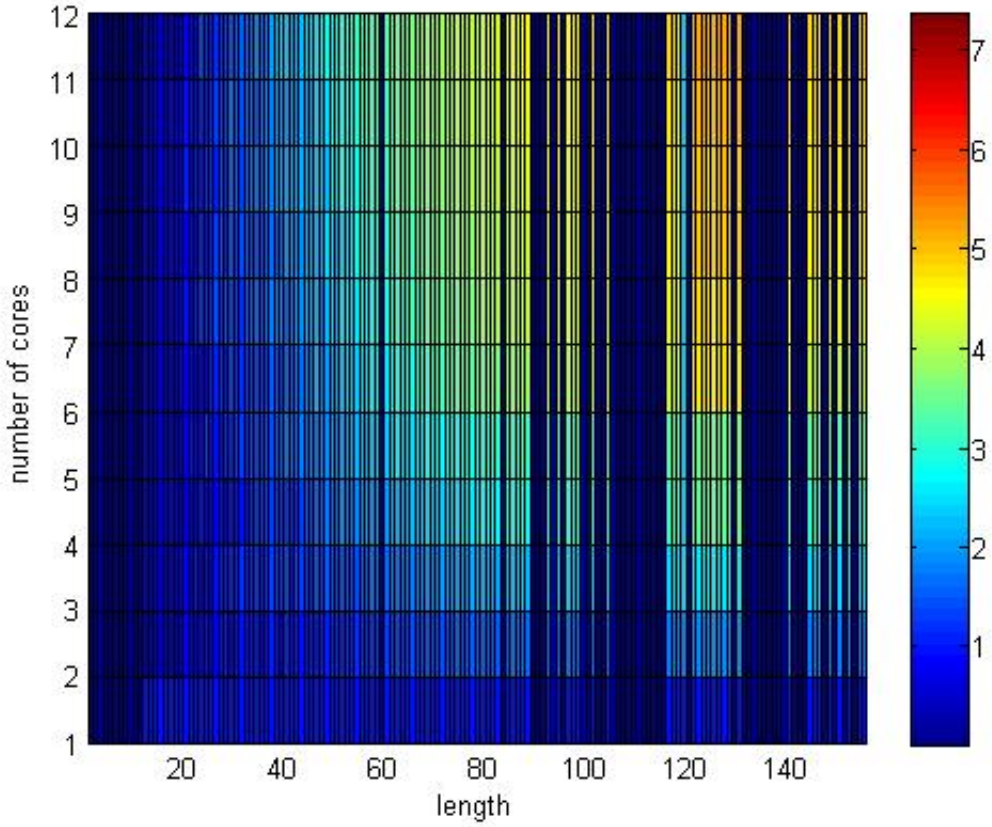
self into shorter runtimes. We also used compiler optimizations to trade off speed vs. size, as amount of RAM is not an issue. The operating system (OS) used was RHEL 6 (Red Hat Enterprise Linux 6).

We used an Intel Xeon system with 64GB RAM and 2 physical CPUs. Each CPU has a clock speed of 2.4 GHz and has 6 independent cores, bringing the total number of processing elements in the system to 12. Each of the CPUs has a 12MB Level 3 cache that is shared by the 6 physical cores. The FSB bus speed is 1066MHz. This server is of type UMA (Uniform Memory Access) and this means that the memory is equidistant from each core and the programs can address and access the entire memory. Therefore, the expected hotspots are the L3 cache hit/miss and also the FSB saturation.

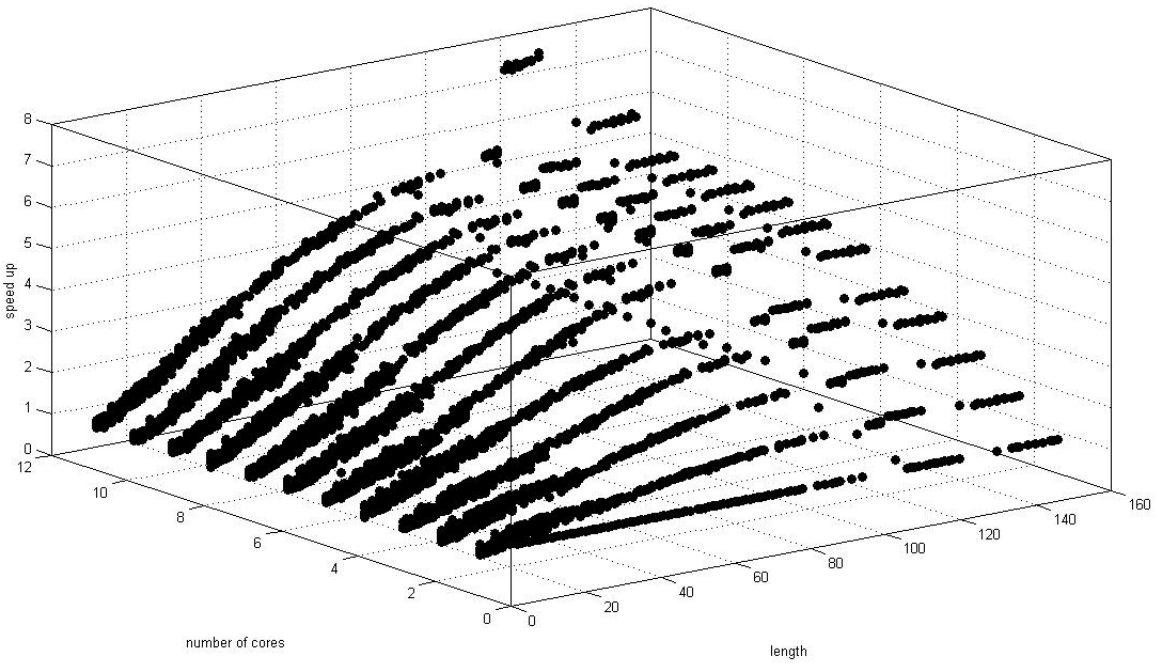
5.6.1 Experiments

The experimental dataset for PKNOTS on the Intel platform comprises of around 700 sequences. The sequences vary in length with the smallest sequence being 16 nucleotides and the longest sequence being 148 nucleotides. All the sequences have experimentally-verified secondary structures and comprises of both pseudoknot and non-pseudoknots. Each of the sequence was executed on 12 cores sequentially and in a step-wise fashion yielding a total around 8,400-runtime data points. Speedup factors for sequences of increasing sequence lengths using varying number of cores are shown as a heat-map in Figure 5.23(a). The information is also plotted as a 3D chart in Figure 5.23(b). From the figures it can be observed that we were able to obtain a maximum speedup of around 6 for a sequence of around 150 nucleotides using all the 12 cores in our test machine.

For small sequences, the relationship between the speedup and the number of core used is not obvious. In other words, for smaller sequences there is no significant performance gain by adding more processing cores. This may be due to several reasons. First, when the sequence length is small, communication overhead between processing cores is more compared to computation time. Second, as the various processes & threads run only for short periods of time cache-misses may be frequent resulting in large amount of clock time spent in memory accesses. As the sequence size increases, this phenomenon fade away as cache-hit increases with computation times becoming more than communication overheads. We observed this trend by using a Linux system tool valgrind (cachegrind) to measure cache-



(a)



(b)

Figure 5.23: Speedup of PKNOTS on Intel x64 as a Heat map & 3D graph

```
==10944== Cachegrind, a cache and branch-prediction profiler
==10944== Copyright (C) 2002-2009, and GNU GPL'd, by Nicholas Nethercote et al.
==10944== Using Valgrind-3.5.0 and LibVEX; rerun with -h for copyright info
==10944== Command: ./pknots -g -k -c -a 4095 SQDfolder/CRW_00562.SQD
==10944==
--10944-- warning: L3 cache detected but ignored
==10944==
==10944== I   refs:      245,505,956,664
==10944== I1  misses:         36,333,152
==10944== L2i misses:            230,094
==10944== I1  miss rate:            0.01%
==10944== L2i miss rate:            0.00%
==10944==
==10944== D   refs:      134,841,062,541  (120,944,888,792 rd   + 13,896,173,749 wr)
==10944== D1  misses:      2,501,408,096  (  2,490,000,906 rd   +     11,407,190 wr)
==10944== L2d misses:        121,615,470  (    119,753,936 rd   +      1,861,534 wr)
==10944== D1  miss rate:             1.8% (            2.0%     +            0.0%  )
==10944== L2d miss rate:             0.0% (            0.0%     +            0.0%  )
==10944==
==10944== L2 refs:         2,537,741,248  (  2,526,334,058 rd   +     11,407,190 wr)
==10944== L2 misses:         121,845,564  (    119,984,030 rd   +      1,861,534 wr)
==10944== L2 miss rate:              0.0% (            0.0%     +            0.0%  )
```

Figure 5.24: CPU Cache-Miss performance benchmark for a sequence of length 68

misses. Figure 5.24 shows the number and percentage of *level 1* and *Level 2* cache misses for a sequence of length 68 nucleotides. It can be seen that cache-misses at *Level 1* is very minimal while at *Level 2* it is virtually not existent. One plausible explanation for the low cache-miss is that parallelized parts of PKNOTS are independent of one another and are cached for longer time in multiple CPU cores. Added to this, is the fact that modern CPUs have larger caches and therefore the caches are not flushed frequently as well.

We now use our speedup measurements across multiple cores to understand the variations in the 'F' values. For this we simplify the Equation in (5.3) to make the variable 'F' as the dependent variable to arrive at the Equation (5.7) with the independent variables being 'n' and 'S'. In this equation 'S' refers to the absolute

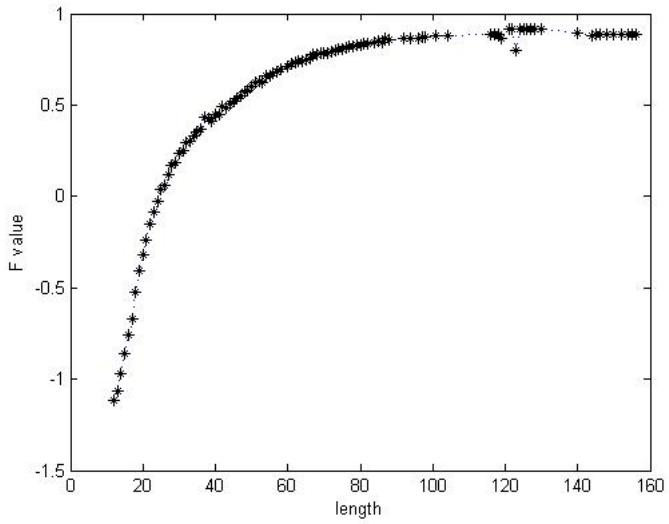


Figure 5.25: F values as a function of Sequence Length

speedup obtained when using 'n' number of cores. We now plot the 'F' values against the sequence length in Figure 5.25. Here we see an increasing trend of 'F' values, as the length of input sequence increases, reflecting the fact that more parts of the program are executing in parallel. Next, we also plot the average standard deviation of 'F' values vs. the sequence length in Figure 5.26.

$$F = \frac{1 - \frac{1}{S}}{1 - \frac{1}{n}} \tag{5.7}$$

From the Figure 5.25 we can see that the 'F' value of PKNOTS for sequence lengths above 120 is roughly 0.9. From this and previous experimental results, we form a hypothesis that '25' is the maximum number of processing cores for PKNOTS. At the number of 25, adding one more processor would only give a gain of 1%. If 'F' = 0.9, speedup will reach the saturation point slightly after 30 processing cores. We were not able to observe the saturation region in our experiments, because the

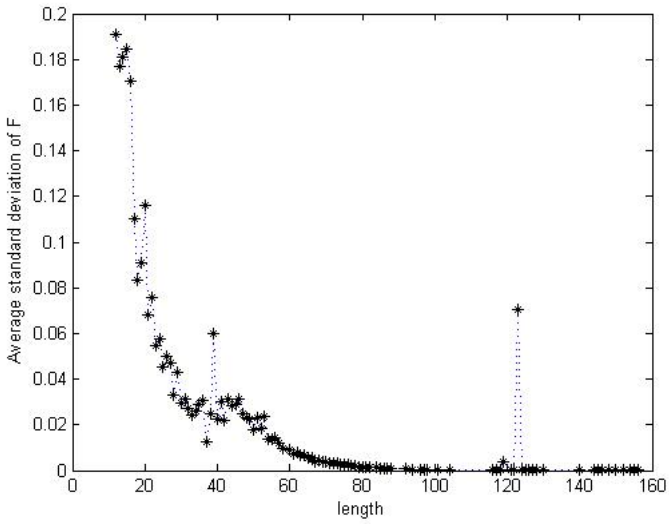


Figure 5.26: Average Std. Dev. of F values Vs Sequence Length

total number of computing cores in our test machine is only 12. Based on this, we are able to provide recommendations on the ideal number of processing cores for sequences with different length. This is plotted in Figure 5.27. This knowledge can be used to do load balancing in a web portal into choosing the right system configuration based on the size of input sequence submitted.

5.7 PKNOTS on Virtualized x64 Architecture

Virtualization is a new technology and enables packing multiple fully functional computing instances, typically servers, into a single physical server. By this way, the hardware resource is better utilized compared to multiple lightly loaded individual systems. While the virtualization technology is being increasingly adopted by the industry to downsize the data centers, we wonder if this technology will be suitable for high performance computing domain. In this section we describe

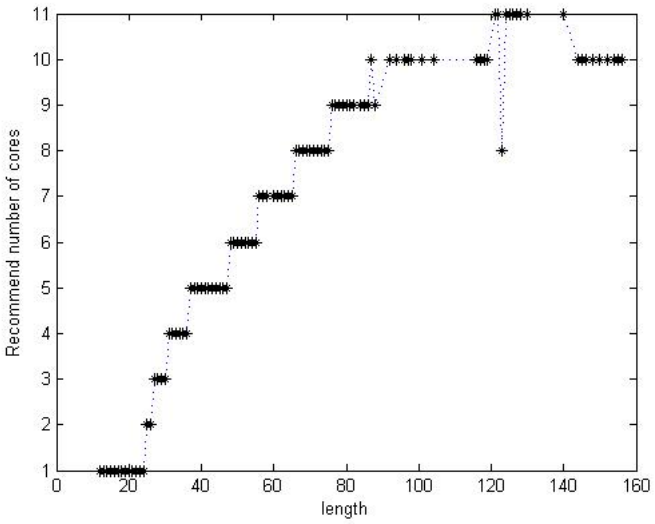


Figure 5.27: Recommended number of parallel cores for various sequence lengths

our experiments with porting PKNOTS to two multi-processor platforms - one of which is a virtual system. The first is a 12-core x86-64 Intel® Xeon® E7450 with 64 GB memory, processor speed 2.4 GHz and 12 MB cache per processor (abbreviated Apollo). The second is a 16-core x86-64 virtual machine with 16 GB memory running on the QEMU Virtual CPU version 0.9.1, processor speed 2.4 GHz and 2 MB cache per processor (abbreviated AVM1). It draws processing power from the Apollo machine. Both x86 platforms run on Red Hat Enterprise Linux 5 with kernel version 2.6.18-164.11.1.e15.

5.7.1 Implementation Method

The POSIX multi-threading library is used to port PKNOTS to Apollo and AVM1. Both the original and parallelized programs are complied using gcc-4.4.1 with O3 optimization.

A total of 140 RNA sequences, drawn from two databases, are used to benchmark the parallelized algorithms. The first consists of 49 sequences found under the PKNOTS software `Demo/` directory provided within the `pknots-1.05` software repository. The second consists of 90 sequences from the on-line repository Pseudobase [12]. Both contain sequences, which are found to have naturally occurring pseudo-knotted secondary structures.

The sequences are analyzed by running the sequential program and the parallelized versions on varying numbers of cores. The command line parameters used for running PKNOTS are `-c -k -t`, which instructs the program to analyze the sequences for pseudo-knotted and co-axial structures with trackback output.

5.7.2 Performance Results & Discussions

(A) Physical Machine - Apollo

The parallel speedup factor of running the entire 140 sequences on the Apollo machine is plotted against the nucleotide length in Figure 5.28. The maximum parallel speedup efficiency attainable (from 2 to 12 processors) are as follows: 102%, 101%, 91%, 90%, 83%, 83%, 83%, 78%, 75%, 69% and 70%. Super linear speedups are attributed toward the effect of combined caches of a multi-processor system.

An inverted-U shape profile can be observed from the results. For sequence lengths below 90, there is an increase in speedup efficiency with increasing length. This

is because longer jobs are offloaded to parallel cores for processing. Therefore, the ratio of synchronization latencies in the total execution run-time decreases for every successive execution, hence a gradual increase in parallel speedup. Likewise, parallel speedup tends to unity for shorter sequences.

The decreasing trend in speedup efficiency, which occurs for longer sequences, is contributed by two factors. Firstly, Dynamic Programming (DP) works by caching previously computed sub problems to compute current sub problems. As such, the number of memory references scales up on the order of $O(n^6)$, equivalent to that of the time complexity. Given a fixed hardware system with limited communication bandwidth between processor and memory, processing longer sequences leads to more memory reference. This increases bus contention that causes processors to halt when the bus is not available for use.

Secondly, due to the nature of the DP algorithm, referenced memory location for consecutive iterations are rarely contiguous. Therefore together with an increase in memory references, these locations cannot fit into the limited memory cache. As such, cache misses increase for increasing sequence lengths, therefore partly contributes to lower speedup efficiency.

Both factors are accentuated when more processors are used, as this would mean a larger amount of stalling time each processor would encounter while waiting for the bus to be available. This can be observed from the gradual development of a 'hump' profile in Figure 5.28 for a greater number of processors.

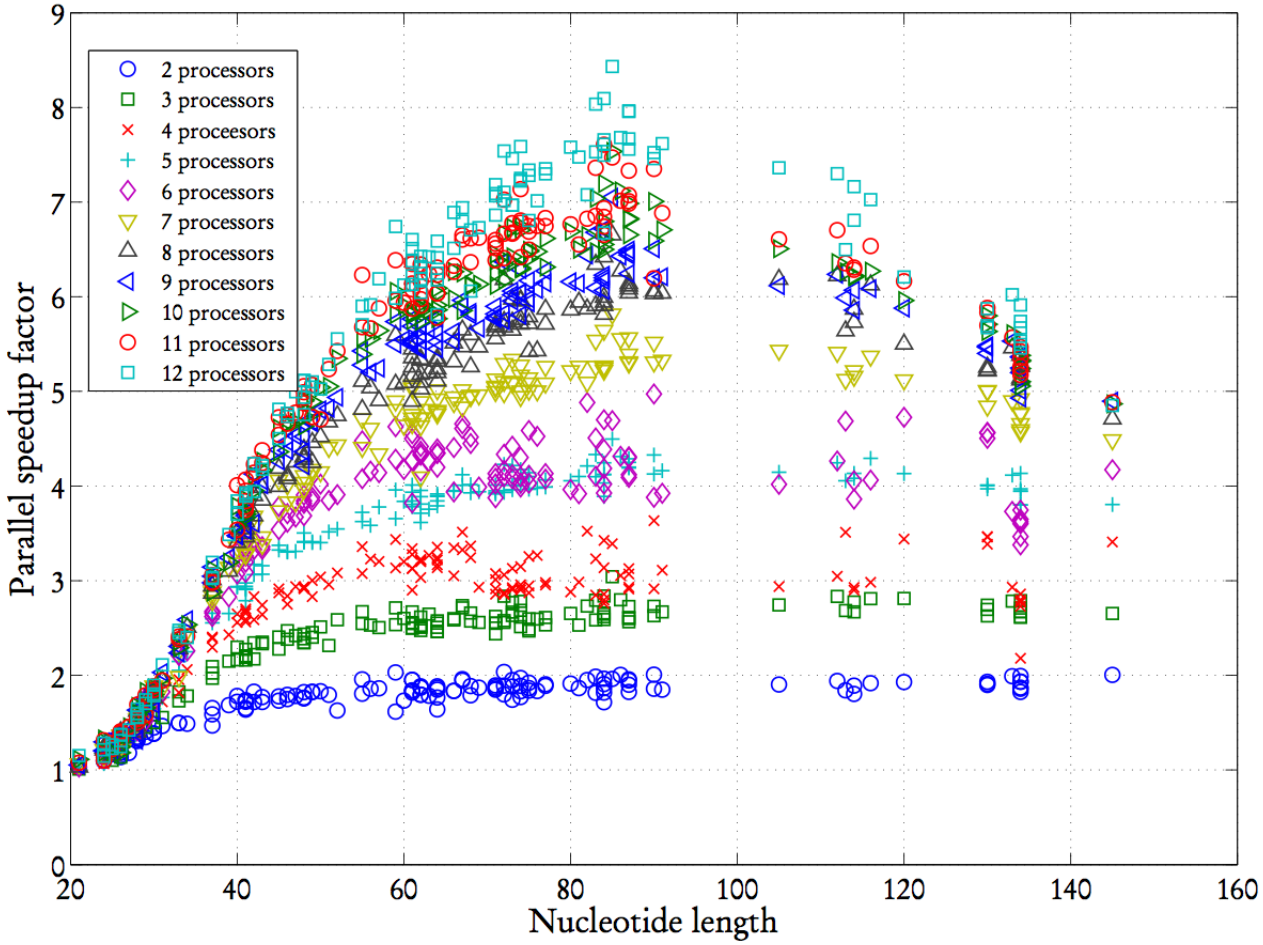


Figure 5.28: PKNOTS Speedup on the physical machine - Apollo

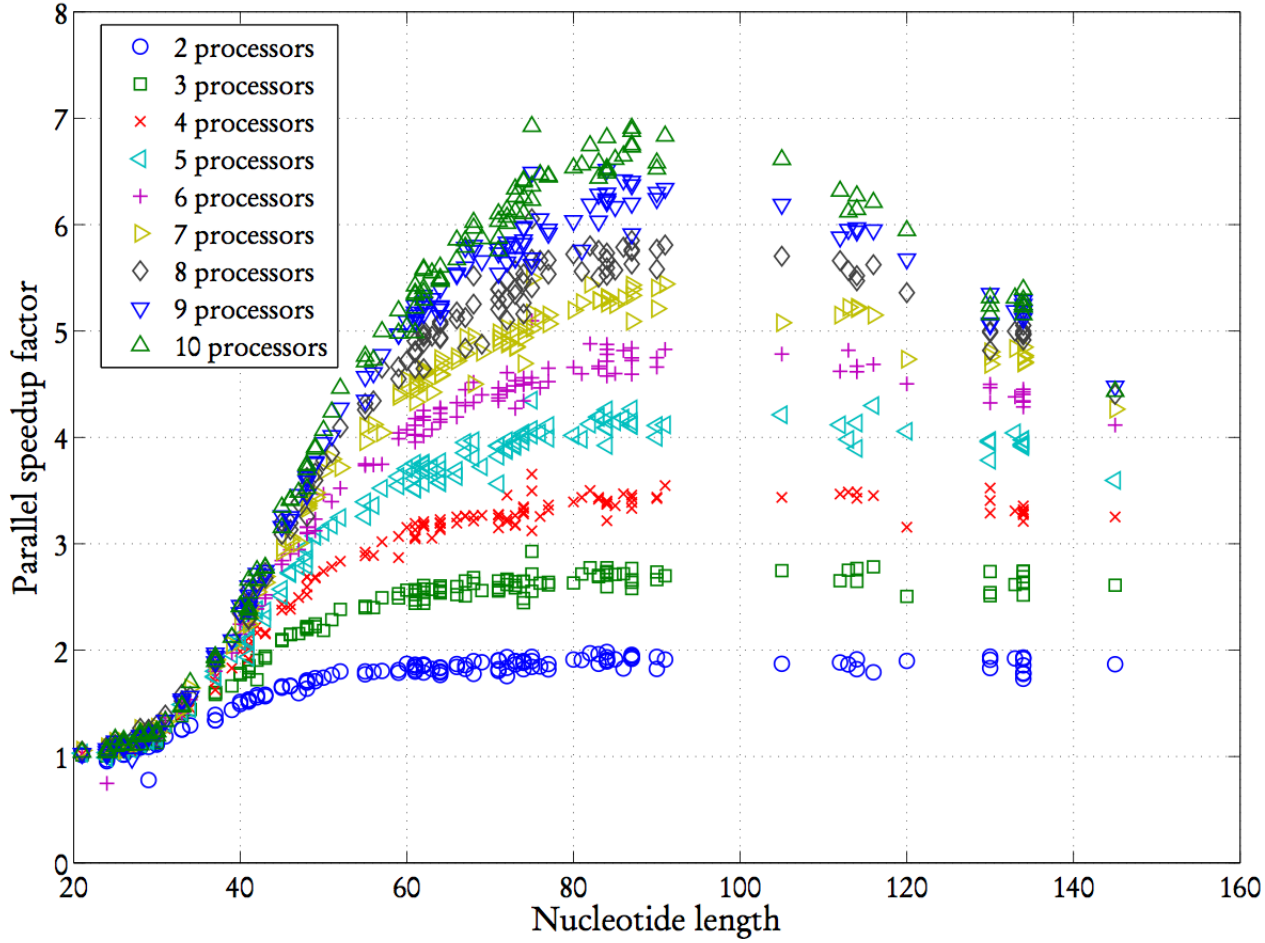


Figure 5.29: PKNOTS Speedup on the virtual machine - AVM1

(B) Virtual Machine - AVM1

The parallel speedup factor of running the entire 140 sequences on the AVM1 machine is plotted against the nucleotide length in Figure 5.29. The graphs of 11 to 16 virtual processors are omitted as their profiles are similar to that of using 10 processors. The maximum parallel speedup efficiency attainable (from 2 to 16 virtual processors) are as follows: 99%, 98%, 91%, 87%, 85%, 78%, 76%, 72%, 69%, 66%, 59%, 54%, 51%, 48% and 46%.

The inverted-U trend is evident from Figure 5.29. Speedup decreases toward unity for short sequences due to the inefficient spreading of workload among the processors.

The overall run-time performance is worse than that on Apollo. One of the reasons is that the per processor cache size for the AVM1 is one sixth of the Apollo. As such, cache misses are higher on the AVM1 compared to Apollo, therefore leading to lower parallel speedup efficiencies. Secondly, the use of virtual machine technology means that there would be overhead incurred in over committing the number of physical cores available for the virtual machine to spread its workload on. Therefore, the performance of using 16 virtual processors is not better than that of using 12 virtual processors since there are 12 available physical processors for use.

5.8 MARSs on Intel x64

In this section, we describe our experiments with MARSs on Intel x64 and discuss the results. We used the same hardware & software environment as described and used with the PKNOTS on Intel x64. The dataset used for the experiments in this architecture contains a total of 726 experimentally verified RNA sequences from various biological databases. Close to 80% of the sequences are from RCSB Protein Data Bank Database and the rest of them are from Nucleic Acid Database, Sprinzl tRNA Database, and Gutell Lab CRW. The sequences are of different sequence lengths and range from the smallest length of 4 nucleotides to the longest length of 545 nucleotides. Most of the sequences have RNA sequence length of less than 80 nucleotides and the average length size is approximately 63 nucleotides. Figures 5.30 and 5.31 show the distribution of lengths of various sequences and also the distribution of sequences across the various databases.

The experiments were done such that the results can be as 'unbiased' as possible. Some of the measures that were undertaken were the use of system primitives such as 'pthread' and 'taskset'. We use 'pthread' to create the required number of threads while 'taskset' is used to pin the threads to different CPUs. Together these primitives will either eliminate or reduce the effects of over-scheduling of the threads to the same CPU(s) by the OS. We used system command 'time' to measure actual time spent by the program and the system. We used our dataset described in the previous paragraph and executed it using multiple cores to un-

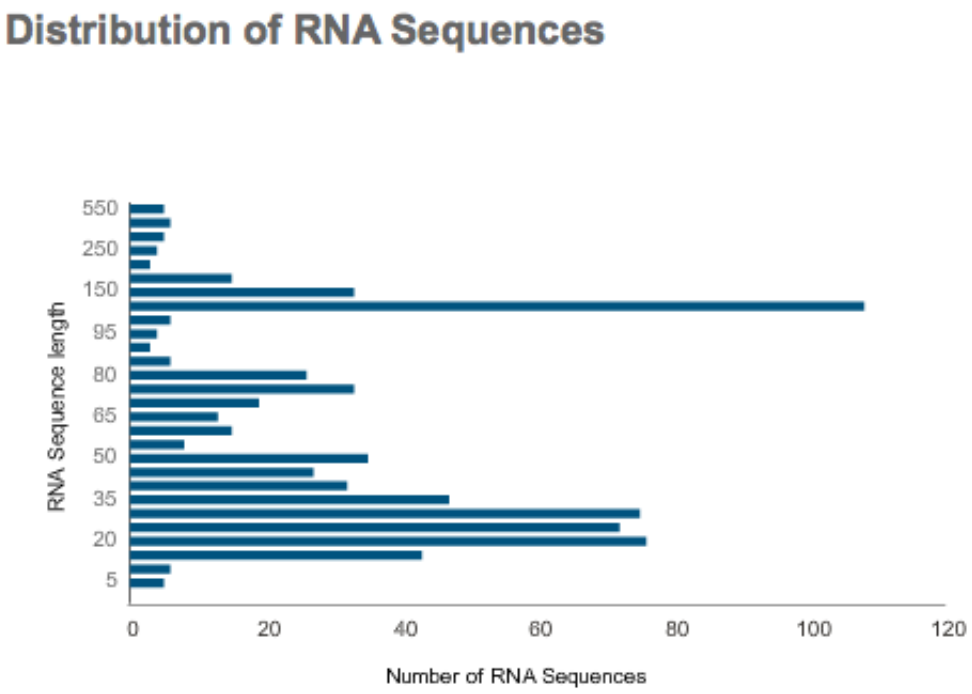


Figure 5.30: Distribution of RNA sequences according to sequence length

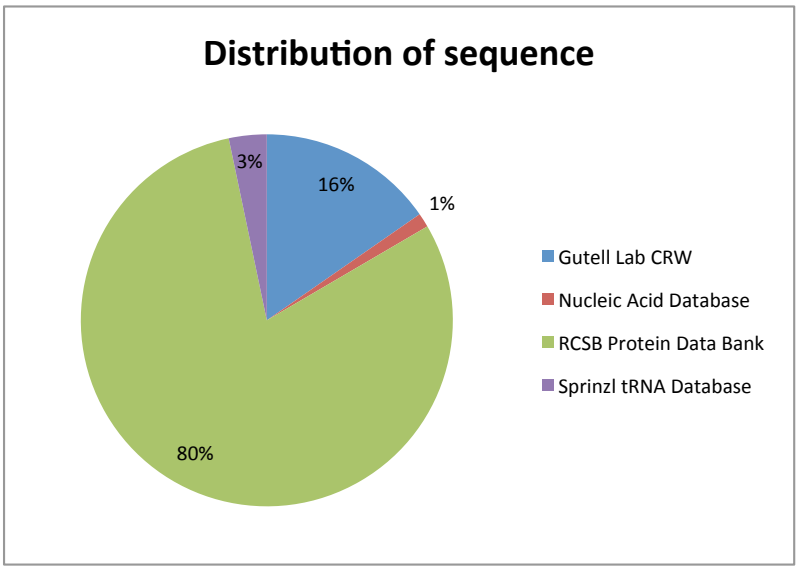


Figure 5.31: Distribution of RNA sequences according to source

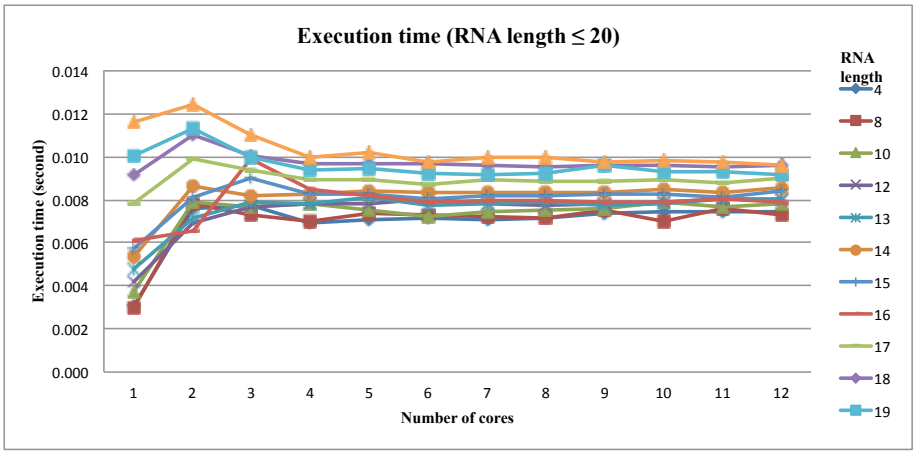


Figure 5.32: Performance of MARSs on Intel - Sequence length < 20 Nucleotides

derstand the performance gain / loss. As expected, for shorter sequences adding more computing power does not increase the performance, instead results in loss of performance. For longer sequences, adding more computational cores results in significant performance gain. Based on our observations we split the results into three graphs as the run-times range from 0.003 seconds to over 12000 seconds. Figures 5.32, 5.33 and 5.34 shows these trends. From Figure 5.32 it can be seen that initially the execution times increases when more computing cores are added before it settles down to a fixed band. This is because adding a second thread in an independent core consumes system time and does not positively contribute to algorithmic runtime.

We again use the two terms - Speedup and Incremental Speedup - to help us to measure the performance gain between a single-core system and an 'n' core multi-core system. To recall, Speedup is defined as the performance of the algorithm in a single-core system vs. 'n' cores. The ideal speedup is 'n' for a 'n' core system and is always measured against the absolute value of '1' or single-core system.

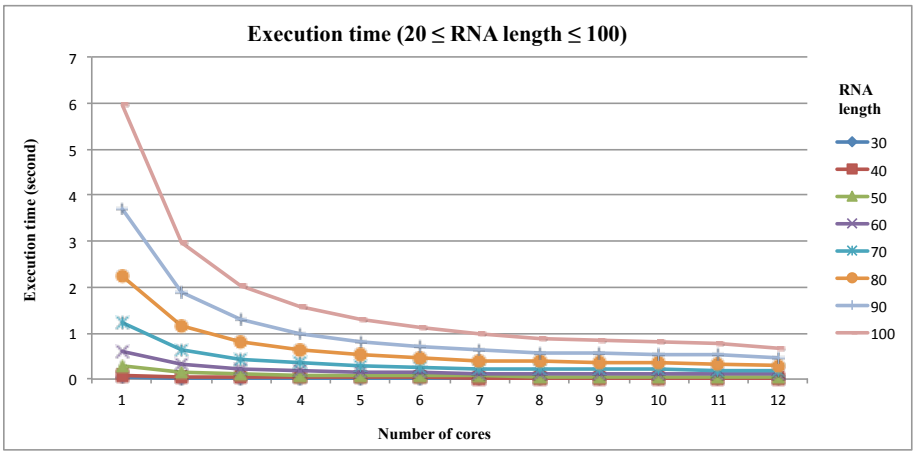


Figure 5.33: Performance of MARSs on Intel - Sequence length (20 < 100) Nucleotides

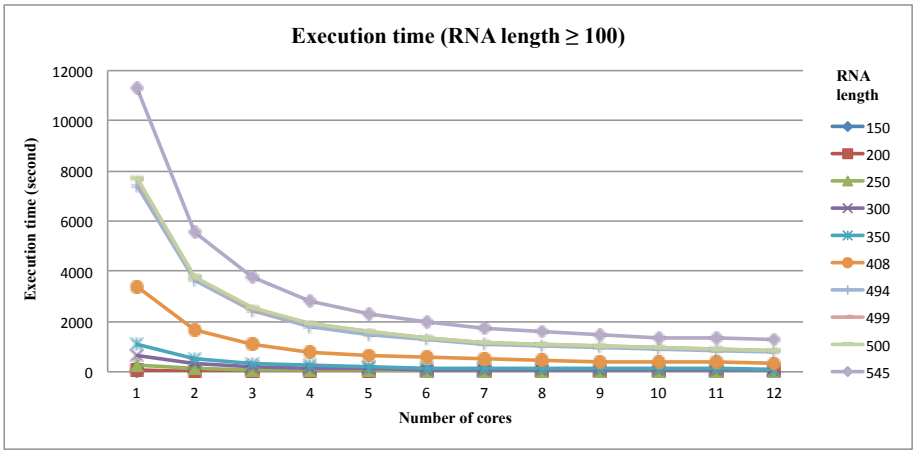


Figure 5.34: Performance of MARSs on Intel - Sequence length > 100 Nucleotides

In contrast, incremental speedup refers to the performance gain that is obtained by adding one more processing element to the existing set of processors. We computed both of these measures for our dataset as given in Figures 5.35 and 5.36. From Figure 5.35 it can be seen for shorter sequences the speedup is low and even negative. This is because the creation of additional thread consumes time but is probably not required or used for such shorter sequences. From Figure 5.36, improvement in execution time decreases as the number of cores increases. This is expected because of the concept of diminishing return, where the overall execution time is improved at a decreasing rate with an increasing number of cores. Another key observation is that marginal improvement is less than 10% or even near to 0% when running at 9, 10 and 11 cores with a sequence length more than 100. This means that sequences running at 8 cores will gain very little or even no improvement in execution time with an additional core. This suggests that it is likely to be optimal when sequences of length more than 100 are running at 8 cores.

In UMA architecture such as Intel x64, it is possible to have task parallelism using either a multi-process or multi-thread model. The difference being in a multi-process model, each task runs in its own address space and does not share data such as global variables by default; privileged system primitives are required for data exchange. In a multi-threaded model all the threads run in the same address space and are able to share data more easily. At the same time, in a multi-process model each process can use up to 4GB in virtual memory size for 32bits. Conversely, in a

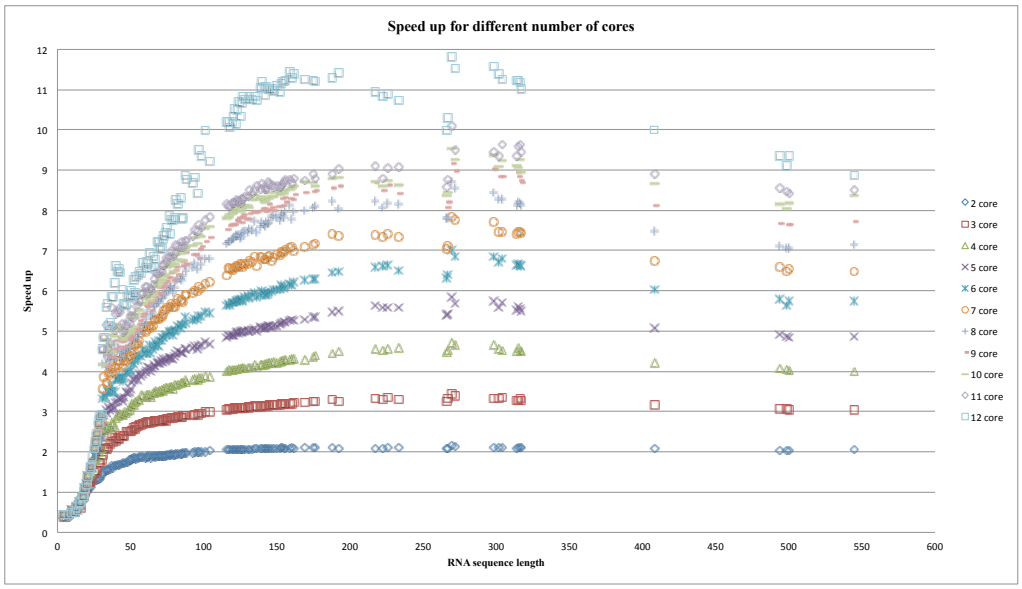


Figure 5.35: Performance of MARSs on Intel - Speedup

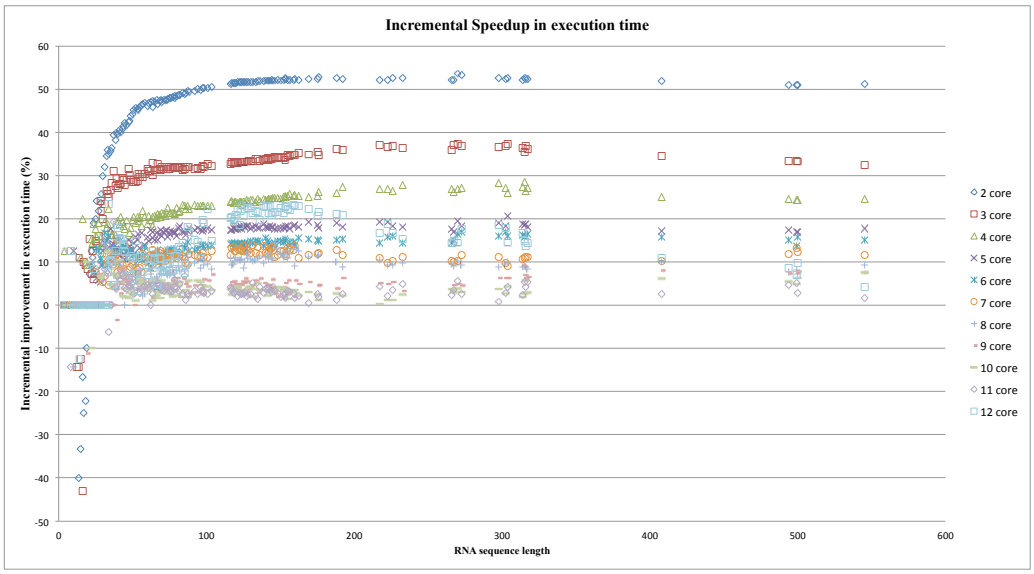


Figure 5.36: Performance of MARSs on Intel - Incremental Speedup

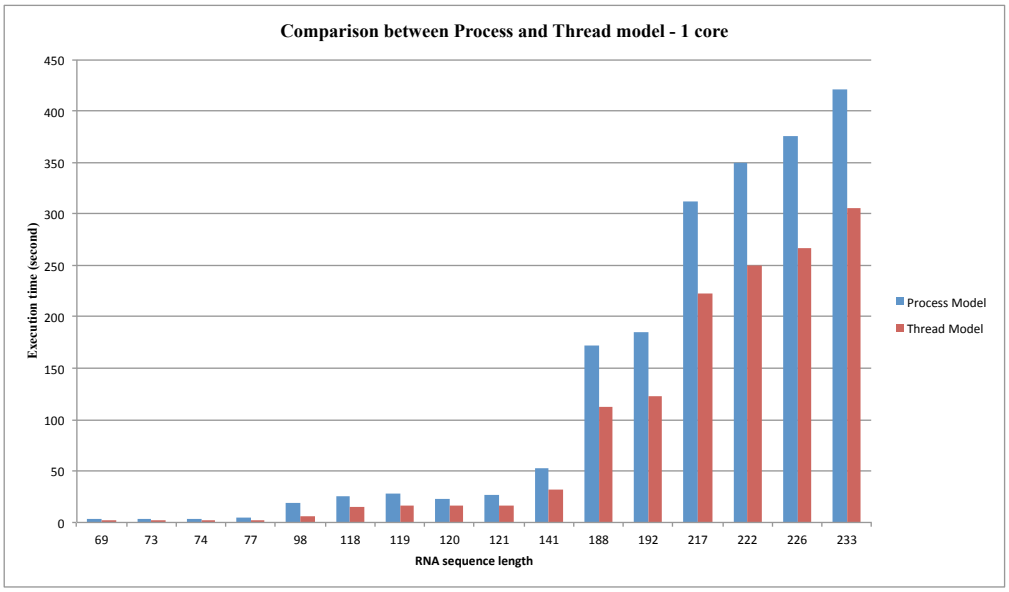


Figure 5.37: Performance of Multi-Process Vs. Multi-Thread Model - 1 core

multi-threaded model all the threads in total use 4GB of virtual memory. In order to evaluate the best model for MARSs we did a simple test using sufficiently long sequences on a single-core and four-core setup. The results are shown in Figures 5.37 and 5.38. Finally, we measure the prediction accuracies of MARSs/Intel using three metrics of PPV, Sensitivity and BP distance as defined in Section 4.11. We used a simple thermodynamic model and the results for the dataset up to sequence of length 63 nucleotides, the average sequence length, are shown in Figures 5.39, 5.40, and 5.41. This shows that MARSs is rather accurate in predicting structures and at the same time can adopt a new & updated thermodynamic models in the future as well.

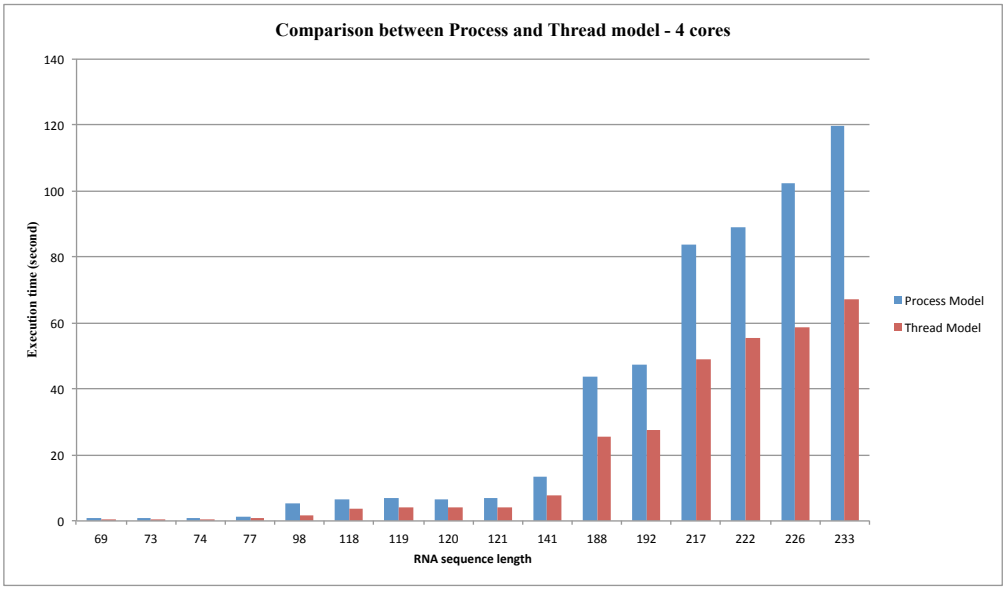


Figure 5.38: Performance of Multi-Process Vs. Multi-Thread Model - 4 core

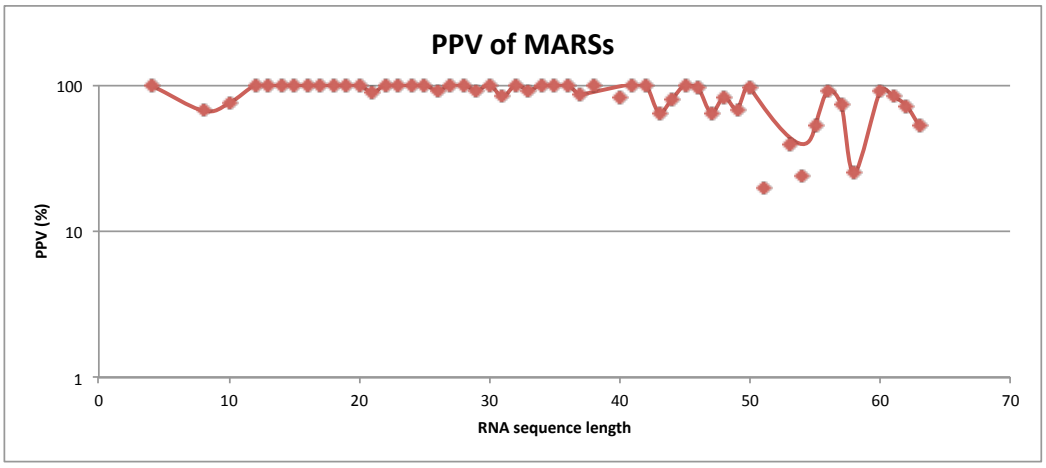


Figure 5.39: Prediction Accuracy of MARSs - PPV

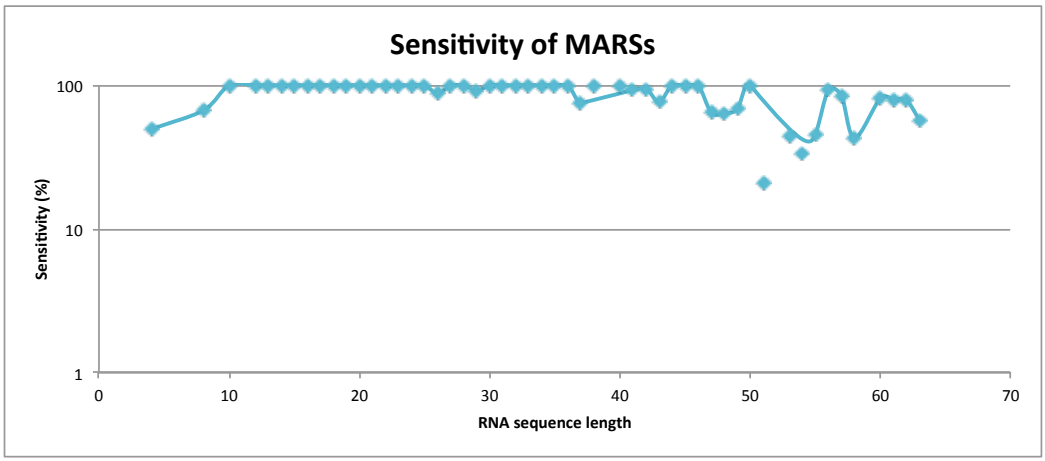


Figure 5.40: Prediction Accuracy of MARSs - Sensitivity

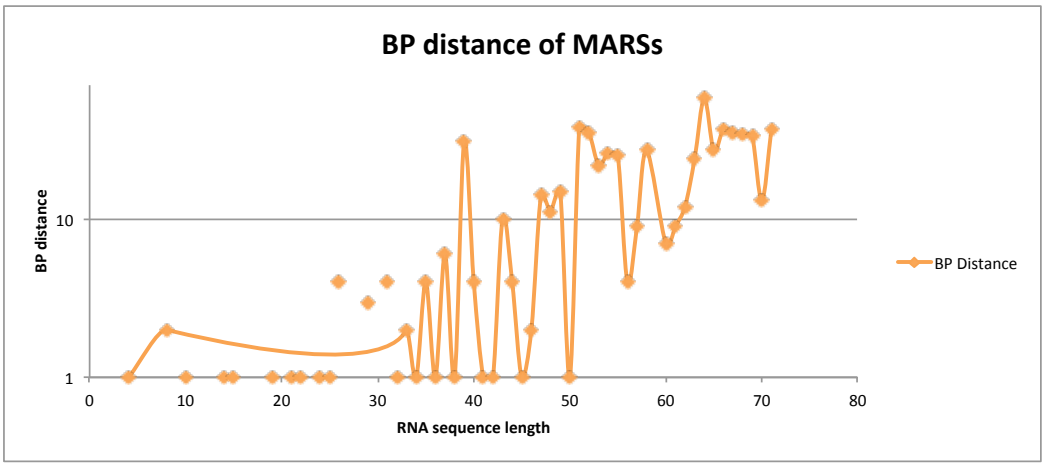


Figure 5.41: Prediction Accuracy of MARSs - Base Pair Distance

5.9 PKNOTS on IBM Cell

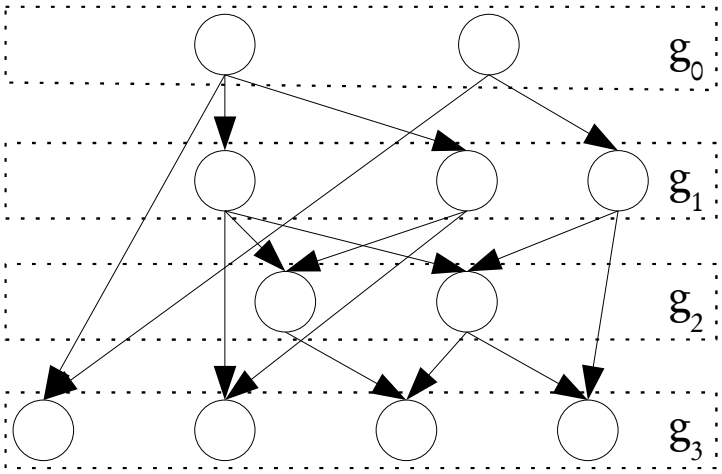
In this section we describe our efforts into parallelizing PKNOTS on the IBM Cell architecture. We use wavefront parallelization technique for this purpose. Wavefront parallelization is a technique for exposing hidden parallelism in dynamic programming algorithms. The formulation analyzes the set of equations characteristic to the algorithm and suggests a suitable parallelization scheme, which allows the algorithm to be ported to a parallel architecture.

DP solves problems by first recursively evaluating their sub-problems. The complete set of sub-problems to be enumerated can be organized as a Directed Acyclic Graph (DAG) shown in Figure 5.42. Each sub-problem is represented by a node in the graph, and every directed edge $A \rightarrow B$ indicates that sub-problem B requires the result of A for computation.

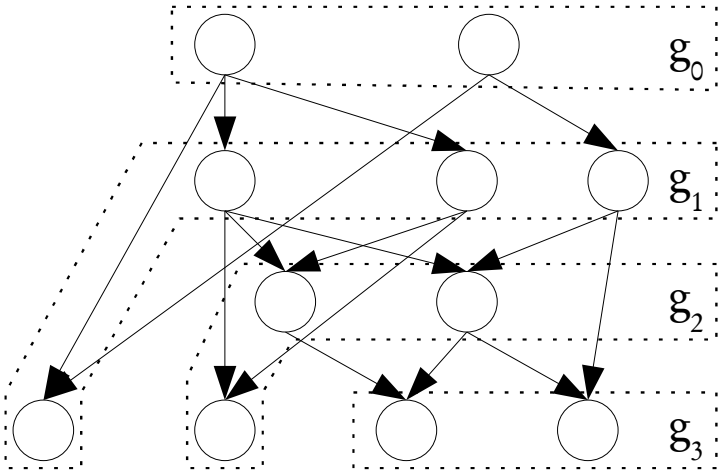
The entire set of sub-problems, denoted S, is a partially ordered set. This implies that any two sub-problems in S can either be computed independently (given that their ancestors[1] have already been evaluated) or not. The latter is only true if one sub-problem is an ancestor of the other.

A parallelization scheme would have to partition S into subsets g_i such that elements within g_i are pairwise independent. Parallelization can then proceed by computing elements within g_i in parallel. Note that there might exist several partitions for S (refer to Figure 5.42).

[1] A is an ancestor of B if there exists a path in the DAG from A to B.



(a)



(b)

Figure 5.42: Two different partitions for a DP problem organized as a DAG

Table 5.2: A partial extract of profiling results running `alphamRNA` through `PKNOTS`.

```
Flat profile:

Each sample counts as 0.01 seconds.
 %   cumulative   self              self     total
time   seconds   seconds    calls  Ks/call  Ks/call  name
70.20  1396.27  1396.27 773517237    0.00     0.00  IntizeScale
26.55  1924.26   527.99   6664923    0.00     0.00  FillWHX
 0.93  1942.67    18.41   6664923    0.00     0.00  FillYHX
 0.84  1959.41    16.74   6664923    0.00     0.00  FillZHX
 0.65  1972.33    12.92      5778    0.00     0.00  FillVX
 0.28  1977.90     5.57   6664923    0.00     0.00  FillVHX
 0.22  1982.20     4.30      5778    0.00     0.00  FillWX
 0.21  1986.41     4.21      5778    0.00     0.00  FillWBX
 0.01  1988.74     0.19         1    0.00    17.49  FillMtx
```

5.9.1 Algorithmic Analysis

The run-time of PKNOTS is analyzed using the GNU Profiler, the results of which are given in Table 5.2. Functions that have single-input single-output behavior (`IntizeScale`) are manually inlined. Indeed, the functions which contribute to the $O(n^6)$ run-time complexity of PKNOTS (`FillWHX`, `FillYHX`, `FillZHX`, `FillVHX`) accounts for the bulk of the computation time. The routines which run in $O(n^5)$ (`FillVX`, `FillWX`, `WBX`) contribute to the remaining runtime.

By parallelizing the code section that consumes the most amount of time, maximum speedup can be achieved. Hence, the parallelization efforts as explained in this chapter shall focus entirely on the gap matrices, that support the cost matrix functions which runs in $O(n^6)$ time.

5.9.2 Hardware Platforms

PKNOTS is ported to two multi-processor platforms. The first is the PlayStation 3 (abbreviated PS3) featuring the IBM Cell Broadband. The PS3 has 1 PPE[2], 6 SPEs[3], processor speed 3.2 GHz and 256 MB memory. The PS3 runs on the Yellow Dog Linux 6.1 with kernel version 2.6.23-9.ydl6.1. The second platform is the IBM Cell Blade Server consists of 2 PPE and 16 SPEs, processor speed 3.2 GHz and 8 GB memory. The server runs on Fedora 12 with kernel version 2.6.31.12-174.2.22.fc12.ppC54.

5.9.3 Implementation Method

The IBM Cell Software Development Kit v3.1 is used to port PKNOTS to the PS3 platform with O3 optimization enabled. Parallelizing the code for the IBM Cell platforms is less straightforward as the SPEs does not have direct access to the main memory. As such, programming requires managing each local store within every SPE, as cache using software. This would mean double buffering, which allows computations and memory transfers to be executed in parallel.

A total of 140 RNA sequences, drawn from two databases, are used to benchmark the parallelized algorithms. The first consists of 49 sequences found under the PKNOTS software `Demo/` directory provided within the `pknots-1.05` software repository. The second consists of 90 sequences from the on-line repository Pseu-

[2]PowerPC Processing Element
[3]Synergistic Processing Element

doBase [12]. Both contain sequences, which are found to have naturally occurring pseudo-knotted secondary structures. Structures with more than 133 nucleotides are omitted from running on the PS3 due to limited physical memory on the platform.

The sequences are analyzed by running the sequential program and the parallelized versions on varying numbers of cores. The command line parameters used for running PKNOTS are `-c -k -t`, which instructs the program to analyze the sequences for pseudo-knotted and co-axial structures with trackback output.

5.9.4 Performance Results & Discussions

Sony PS3

The average speedup factor of analyzing the nucleotide sequences on varying numbers of SPEs are shown in Figure 5.43. The maximum speedup efficiency attainable (from 2 to 6 SPEs) are as follows: 97%, 92%, 86%, 80% and 72%.

It can be seen that wavefront parallelization yields reasonably good speedup efficiencies for the range of nucleotide lengths used in our experiments. This is because the architecture features an efficient EIB[4], which allows memory access to be serviced at a high rate. Therefore, this architecture does not suffer too much from excessive bus contention, which is evident on the x86-64 machines.

[4]Element Interconnect Bus

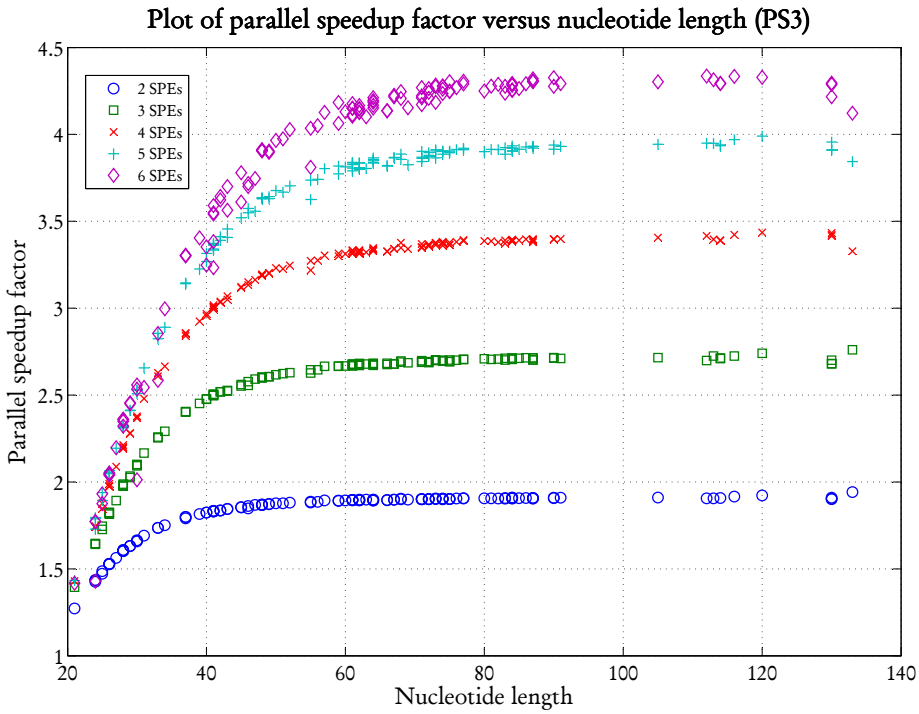


Figure 5.43: PKNOTS speedup graph on the PS3 machine.

PowerXcell 8i Blade Server

Unlike the inverted-U phenomenon that is found on the x86-64 system, the performance of the parallelized PKNOTS does not suffer from bus contention for long sequence analysis. This is attributed to a highly efficient EIB, which allows fast memory service rates. Figure 5.44 shows the parallel speedup graph on the blade server.

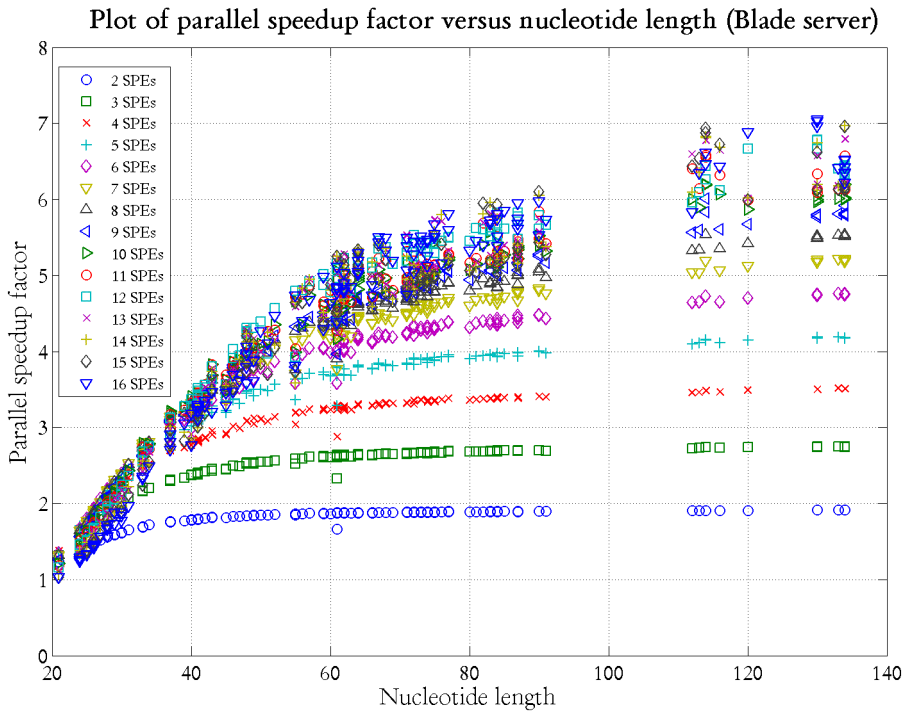


Figure 5.44: PKNOTS speedup on the Blade server.

5.10 MARSs on IBM Cell Broadband Engine

In this section, we describe the efforts we have put in developing MARSs on IBM Cell Broadband Engine architecture. IBM Cell is heterogeneous processor architecture (and unlike our other two architectures) needs special system-level software development to take full advantage of the architecture. More specifically, the Power Processing Unit (PPU) and Synergistic Processing Unit (SPU) are based on different Instruction Set Architectures (ISA). Therefore, for any routine to execute in both of the processing elements, it needs to be compiled differently. At the same time, it is a standard & good practice to run the administrative code in the PPU while executing the CPU-bound code in the SPUs. Towards this end, we archi-

tected our application so that sub-routines that enumerate base-pairs - symmetric folding, asymmetric folding first, asymmetric folding best - are executed on the SPU while the *Level 1* and *Level 2* root tasks along with watch dog monitor tasks execute on the PPU.

5.10.1 Handling Space Complexity

Our first challenge in bringing MARSs to Cell is to adapt the implementation to fit the constraints of the SPU. More specifically, SPU has a limited local storage of size 256KB. This storage space is to be shared between code and data. In MARSs, the size of the *Affinity Matrix* is dependent on the primary sequence length 'n' and therefore either we have to limit the size of the input sequence or adapt the implementation to accommodate this restriction transparently. Without considering any code, the max length of primary sequence can be 512. We instead have designed and implemented an auto-tiling scheme that makes available necessary parts of the affinity matrix to the SPUs. In this scheme, the affinity matrix is split into square tiles of fixed size. In our implementation, we choose a tile size of 32x32. Using 64bits per value this could use up to 8K of memory leaving out the rest for code and program data. However, as more tiles are transferred from SPUs to PPEs this could result in increased latency and leave SPUs idle. Therefore, we implemented a second efficiency-improving scheme that would use less space and therefore save bandwidth. Each of the matrix elements that is 64 bits in size is replaced with 4 bits pointing into the lookup table of base-pair matrix for corre-

sponding base-pair affinity values. By this way, the total space required for a tile of 32x32 dimension is only 0.5K instead of 8K. This data size reduction scheme works well for our simple thermodynamic model. Other thermodynamic models that have more than 16 values can be indexed with higher number of bits, say 8 bits that can index 256 values. Using the above two methods, we are able to effectively work around the SPU memory size constraints. These methods can be classified under data parallelism.

5.10.2 Handling Task Parallelism & Scheduling

IBM Cell is a multi-core processor like Intel Xeon that we have used in our experiments. At the same time, unlike Intel Xeon where each core has full accessibility to the main and secondary memories, Cell's SPUs cannot directly access either of the memories and the I/O interface. This architecture requires DMA (Direct Memory Access) to be performed to load both the code and data to each of the SPUs. As the number of SPUs can vary between systems - 6 in Sony PS3, 8 in Cell and 16 in a PowerXcell blade server - there is a variable amount of latencies when using round robin scheduling with barrier synchronization. For most practical applications, this latency is not an issue due to the presence of high-speed communication bus EIB (Element Interconnect Bus) that connects all the SPUs and PPU. We endeavor to test the performance of Cell for our HPC algorithm MARSs.

MARSs has been structured so that the *Level 1* and *Level 2* root tasks act as an

admin and runs in the PPE. The 3 base-pairing routines are compiled as individual SPE programs. During startup, MARSs using system commands finds out how many SPUs are accessible to itself. This is then used in the scheduler routine to schedule concurrent tasks to the SPUs in a round-robin fashion. MARSs also slices up the affinity matrix into tiles of 32x32 dimensions and transfers the required data along with the code to the SPUs. During one execution cycle, the admin task in PPE provides each of the SPEs with an address in main memory from which to fetch the code and data; each of the SPUs then initiates memory transfers to themselves. We found this model to perform better instead of the PPE providing each of the SPEs required information in a single-threaded fashion. After the computation, each of the routines transfers back the results to the main memory and notifies the PPU of task completion. It can be observed we use shared memory interface for synchronization of tasks. There is also a need to use barrier style synchronization as each of the SPUs potentially can be working on different tasks. For example, we deploy 3 different base-pairing routines to 3 SPUs and then have to wait for all the results before deciding the best base pairs. Therefore, while the PPU task is waiting for all the SPUs to finish, there is an unavoidable & variable amount of idle time at each of the SPUs. It is apparent that we use task parallelism as well.

5.10.3 Performance Results & Discussions

Our experimental dataset in Cell comprises of around 1000 RNA real sequences from various RNA databases with known secondary structures containing both pseudoknots and non-pseudoknots. The sequences in the dataset are from few nucleotides to few hundreds nucleotides giving us the size diversity as well. Each of the sequences were tested using 1 to 16 SPUs for performance measurements resulting in 16,000 data points. Our first analysis is to understand the performance gain (or loss) by using multiple SPUs for sequences of diverse varying length. For this experiment, we used a subset of sequences with sequence lengths 8 to 154 nucleotides in length. As the performance metrics varies from a few seconds for shorter sequences to a few hundred seconds for longer sequences we split the chart into two, one for sequences below 32 nucleotides that show different runtime pattern and one above 32 nucleotides that shows a different runtime pattern. These are shown in Figures 5.45 and 5.46. As can be seen in Figure 5.45, the performance for shorter sequences actually falls i.e., runtime increases on adding more SPUs to the processing pool. This is because when there are SPUs the scheduling sub-system has to wait for all of them to finish before a new set of jobs can be assigned. As the length of the sequences is rather short, the SPUs finish the tasks quickly and are idle for longer time periods. More specifically, the communication, synchronization, and administrative overhead is more than the performance gains for shorter sequence with more SPUs. In contrast, in Figure 5.46 it can be observed that for sequences longer than 32 nucleotides there is positive performance

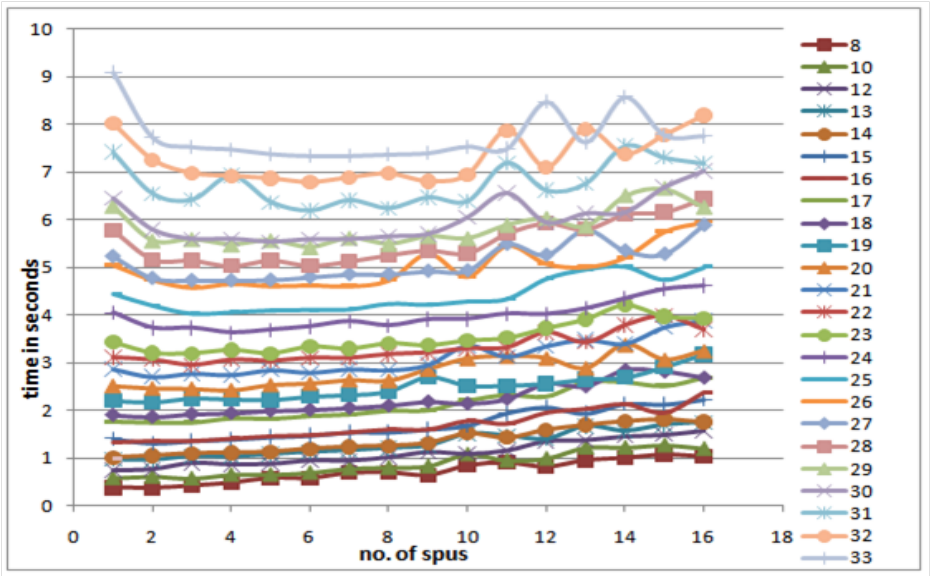


Figure 5.45: Performance of MARSs on Cell for sequence lengths < 32

gain i.e., shorter run-times as the sequence length increases. In this case, the performance gain is more than the total overheads and we witnessed the trend to continue for the rest of the sequences in our collections. Using these runtimes we computed the speedups for sequences with lengths above 32 from 2 SPUs to 16 SPUs. The speedups are shown in Figure 5.48 and from the figure it can be seen that the speedup varies between 2 and 13. The dataset comprises of sequences from 32 nucleotides to 320 nucleotides.

Next, we measure the amount of time the PPU is idle. We define the 'PPU idle time' to be the sum of times within all execution cycles the PPU is idle. This includes the time when the PPU (after assigning a task to SPU) is waiting for SPU to do DMA transfer, when the SPU is performing the task and PPU waiting for all SPUs to finish the assigned tasks before another round of tasks can be assigned. We instrumented the binaries to compute the values and again split

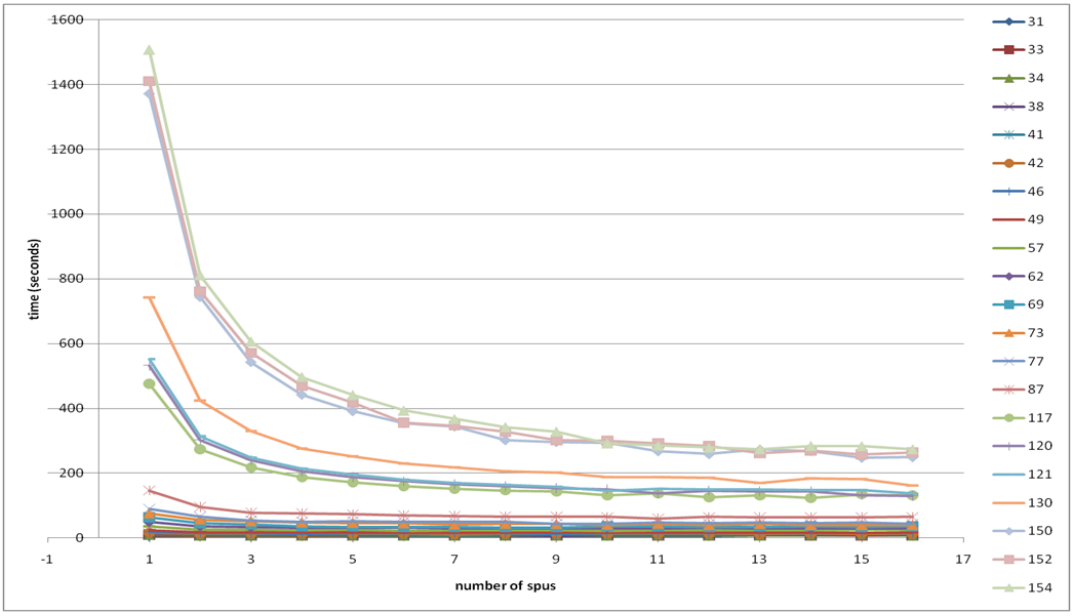


Figure 5.46: Performance of MARSs on Cell for sequence lengths > 32

the graph into two - for sequences less than and greater than 32 nucleotides. We take representative samples within these two categories and show them in Figures 5.47 and 5.49. From Figure 5.47 it can be observed that for shorter sequences the PPU idle times initially falls and then rapidly increases. This hints at an optimum number of SPUs for a sequence of given length. In general, it can also be seen that a sequence with length 19 has a longer idle time in PPU compared with a sequence of length 8 nucleotides. In contrast, Figure 5.49 shows the PPU idle time for sequences from 32 nucleotides in length to 192 nucleotides. It can be seen that the PPU idle times follows a rapid downward exponential curve as the length of the sequence increases. We have also computed the percentage of the PPU idle time in the total runtime for sequences of various lengths and for different number of SPUs. This helps us to understand the actual amount of time the PPU is idle for MARSs on IBM Cell. This is shown in Figure 5.52. From the figure, many

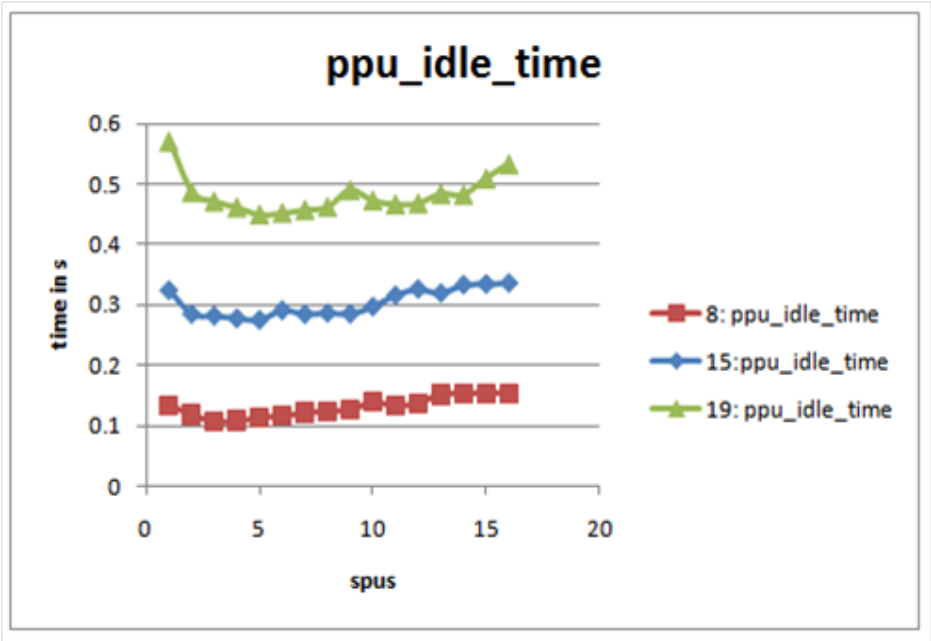


Figure 5.47: MARSs on Cell - PPU Idle Time for Sequence Lengths < 32

interesting trends can be observed. First, the percentage of PPU idle time is never more than 1% of time for all the sequences that we have tested and this is indeed good. Second, for each of the sequences, the idle time monotonically decreases as the number of SPUs increases. Third, the percentage of idle time monotonically increases for longer sequences for every SPU configuration. Overall, the nominal value of 1% idle time is acceptable.

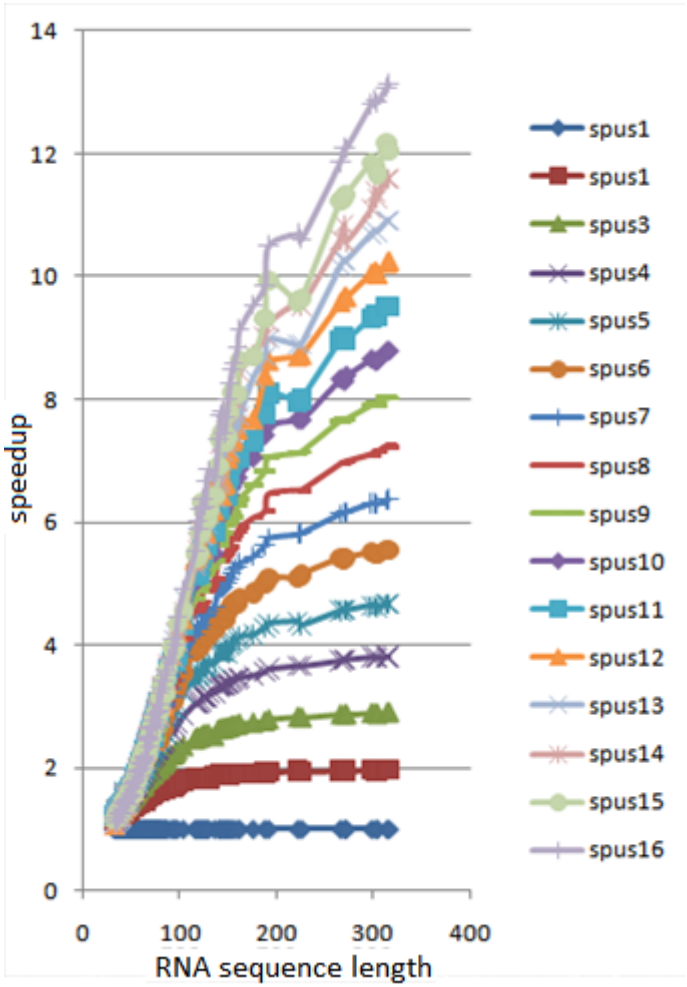


Figure 5.48: Performance of MARSs on Cell - Speedup

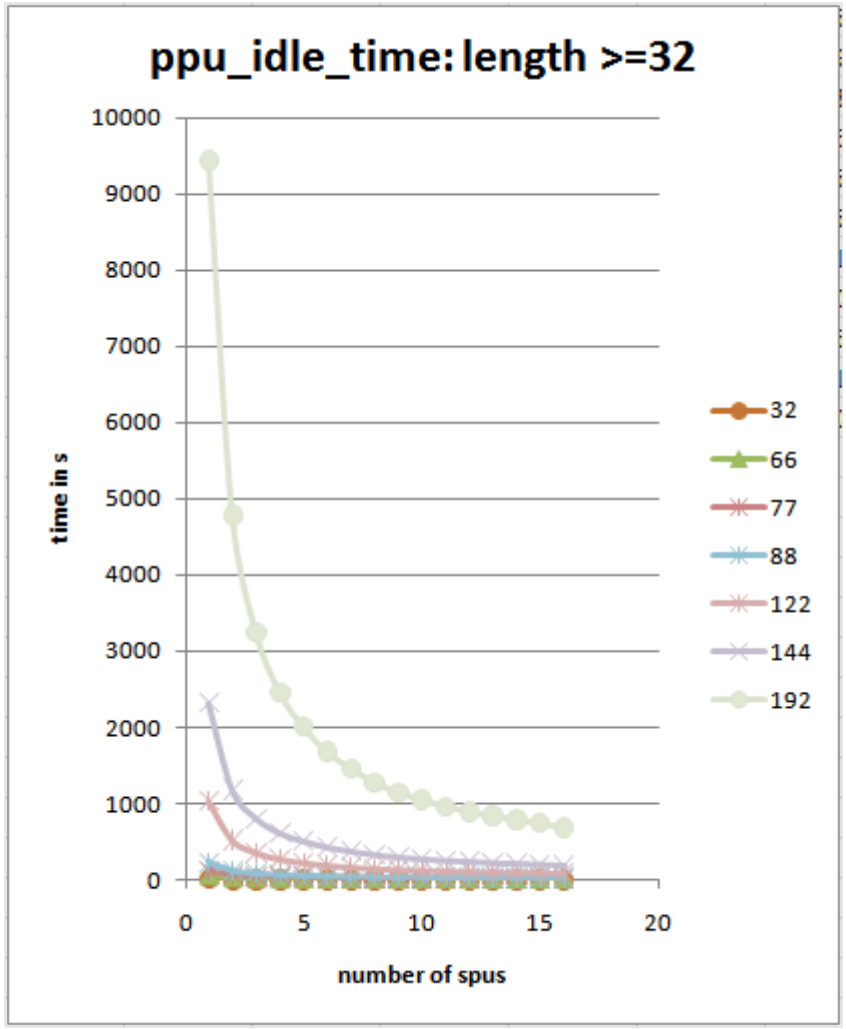


Figure 5.49: MARSs / Cell - PPU idle time for seq. len. > 32

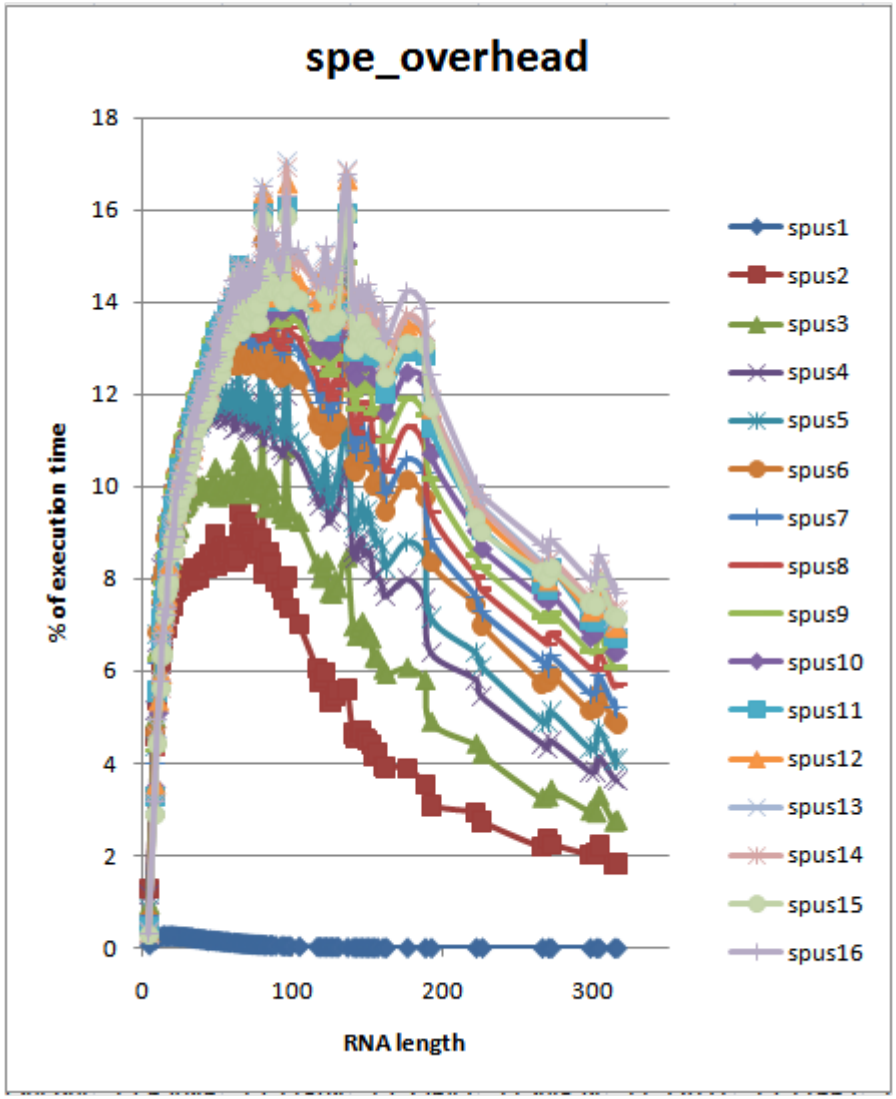


Figure 5.50: MARSs on Cell - SPU Overhead Time

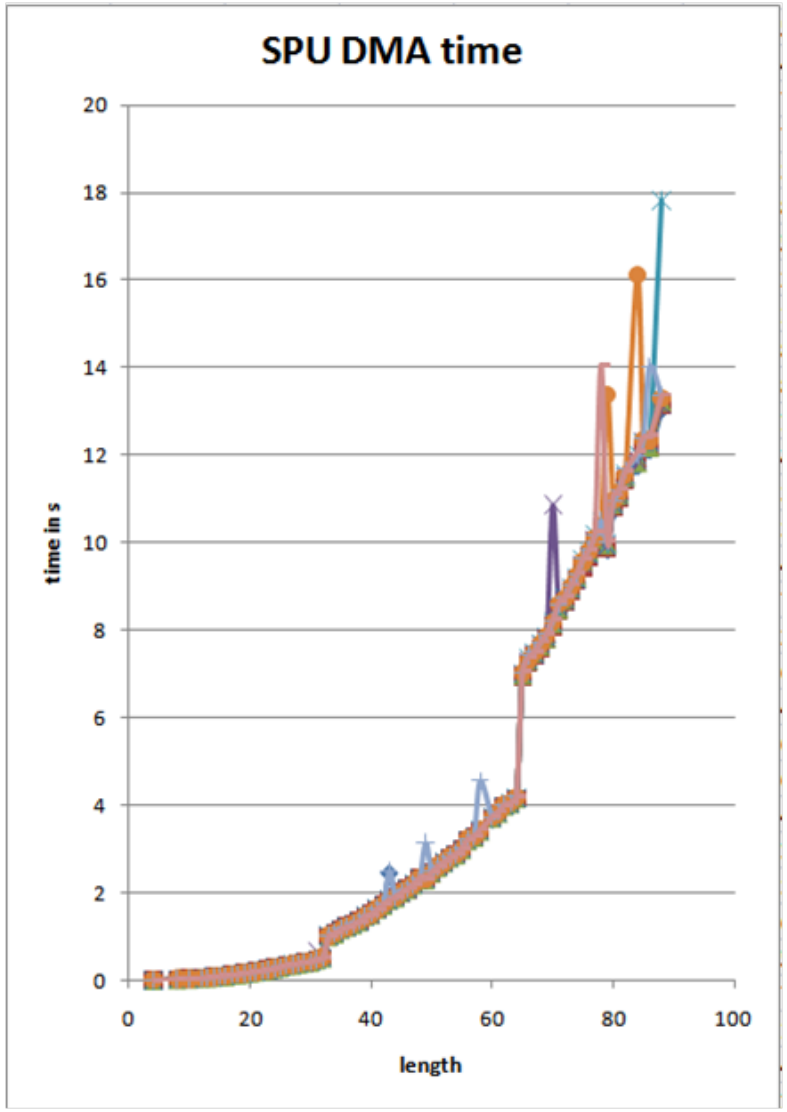


Figure 5.51: MARSs on Cell - SPU DMA Time

Following this, we examined the wait times from the SPU perspective. We call this SPU overhead and is the summation of idle times when the SPU is not doing any task. For shorter sequences & fewer SPUs the total idle time is less and is more for more SPUs & shorter sequences. For longer sequences & few SPUs the wait time is lower but the runtime will be higher as well. For longer sequences & more SPUs the idle times will be less. We used a dataset of sequences with sequence lengths up to 300 nucleotides and measure the SPU idle times. This data is shown in Figure 5.50. It can be seen that a distinct hump pattern occurs for sequences of increasing length and using progressively more SPUs. We also measure the amount of time spent from the SPU side to fetch the affinity matrix slices. This is shown in Figure 5.51 and from the figure it can be seen that the time spent increases with longer sequences. In addition, a distinct pattern occurs periodically. This is linked to the tile size of 32x32 and repeats at multiples thereof.

Finally, we measured the performance of MARSs from a quality perspective and the results were the same as with MARSs on Intel as given in Section 5.8. Therefore, we choose not to repeat it here.

5.11 Inferences from our Performance Evaluation Studies

In this section, we discuss on the various trends from our six experiments conducted as part of this research study. The objective is to summarize & discuss the results

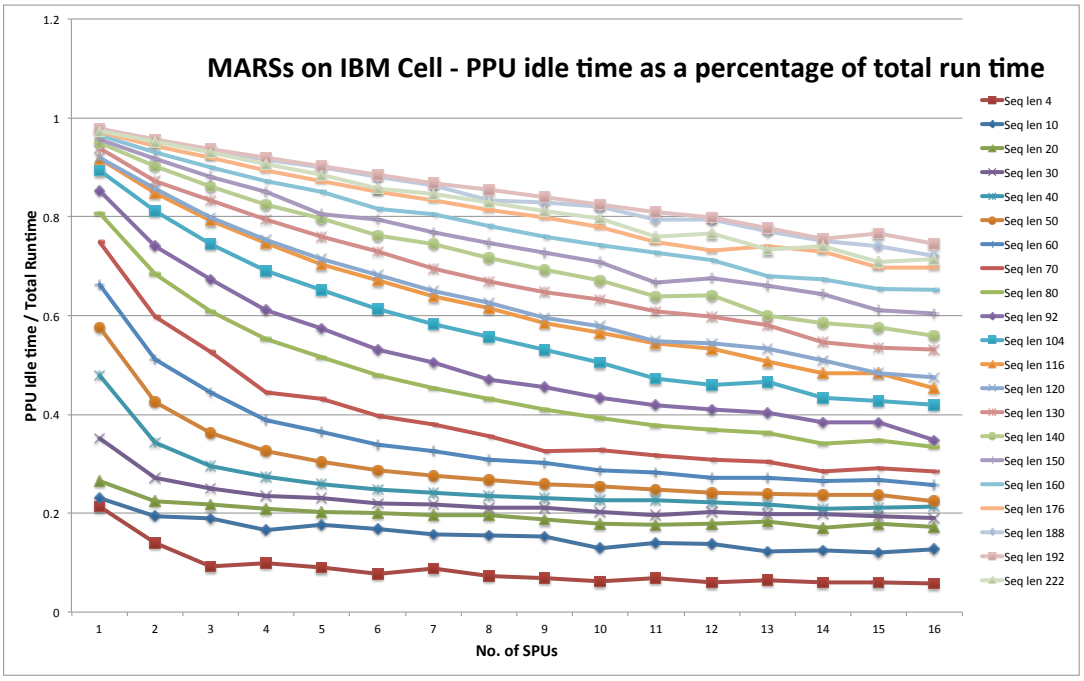


Figure 5.52: MARSs on Cell - Percentage of PPU Idle time / Total Runtime

from the various experiments we have conducted.

Dynamic Programming based algorithms such as PKNOTS, when parallelized efficiently, benefits from executing on auto-scaling platforms like Google App Engine (GAE). The advantage of using GAE is that it is able to scale to input sequences of arbitrary length, by automatically allocating additional parallel compute instances. At the same time, in the current version, the time taken is on the higher side. We believe that this is just a transition as more efficient system primitives are being developed on the GAE platform. These include background instances, faster instances, higher RAM allocations, workflow APIs, and mapreduce APIs. Using these newer methods, we believe the runtimes can be reduced by a large extent. We have also shown that algorithms that are parallelized-by-design, such as MARSs, is able to scale on the GAE easily and produce better results, in the

same platform release that we tested PKNOTS. This emphasizes the point that all new algorithms should consider scalability as part of the design process rather than as an after thought.

From our second set of experiments on the Intel x64 architecture, we can see both similar (as with GAE) and distinct trends. The computational speedups of DP based algorithm PKNOTS is significantly lower than the speedups for parallel-by-design algorithm MARSs. This is similar to our observation from the GAE experiments, where the runtimes of MARSs was superior to PKNOTS. The distinct trend that we observe here is that even with a faster CPU (when compared to GAE instances CPU speed), the performance could not be raised significantly. The performance bottleneck is in other parts of the system architecture, like multi-level caches that result in frequent cache flush & fetch and bus speeds & availability. The main observation being that the limited system resources (CPU, RAM) limits the length of input sequence that can be processed.

From our third set of experiments on the IBM Cell architecture, we can again see similar trends with Intel x64 architecture and unique trends. The speedups of the PKNOTS algorithm is generally less than the MARSs algorithm and this is expected. At the same time, we don't see the bell or hump curved that we observed on the Intel architecture. We believe this is because of the superior system architecture of this platform and the presence of both high-speed interconnecting bus and a dedicated DMA controller. All these system capabilities computed together resulted in better performance for both MARSs and PKNOTS with the

former gaining more.

In summary, in this chapter we present extensive results from our large-scale experiments. We infer that an auto-scaling architecture with practically infinite resources is promising and will be better suited for HPC-type workloads with at least the recommended updates. We also infer that a specialized architecture like IBM Cell outperforms a generic architecture like Intel x64 for HPC class workloads. However, limited system resources in both of these systems limits the size of the inputs that can effectively be removed if the algorithms are designed in such a way that they only process a part of the input rather than the entire problem.

Chapter 6

Conclusions and Future work

In this chapter, we conclude our research and summarize the contributions described in the earlier chapters. We also provide some thoughts for future enhancements and improvements.

The computing capabilities offered by High Performance Computing (HPC) systems are being effectively used to solve complex scientific problems in multiple domains including the bioinformatics domain. The large-scale nucleotide sequencing initiatives undertaken by many biological labs across the world is producing more and more DNA/RNA sequence data. Determining the secondary and tertiary structures of all these sequences using biophysical methods is prohibitively expensive (time & cost). Therefore, there is a strong need for computational algorithms to play a leading role in predicting the higher-order structures of the sequences.

During the last decade of the 20th century the performance scaling of the structure

prediction algorithms was largely dependent on the performance gains of the single CPUs. During the last decade however, chip manufacturing has hit many physical limits such as thermal wall, memory wall and frequency wall. Due to this, the performance gains in the new generation of CPUs are no longer being realized through higher CPU frequencies; instead they are being realized through parallel computing i.e., increasing the number of parallel computing cores on a single die and adding multiple processors. At the time of writing this thesis, the maximum number of processing cores in a general-purpose CPU is 8 (AMD Bulldozer) while it is 512 on a GPU (Nvidia Fermi). Existing computational algorithms needs to be redeveloped (not just ported) to take full advantage of this new-generation of HPC systems. In addition, newer computational algorithms that are multi-core aware by design needs to be developed. Along with the multi-core CPU revolution, another upcoming & promising computing paradigm is cloud computing.

This trend of both multi-core and cloud computing required an in-depth analysis of algorithmic performance of both existing and new algorithms. Therefore, we believe the contributions of this thesis to be timely and useful to the scientific community who are either evaluating the different architectures for use in their studies or who need to know the performance benchmarks and therefore the system limits of one of the architectures that we have studied in depth.

There are two primary contributions of this thesis. First, we parallelized an existing RNA secondary structure prediction algorithm in three different HPC architectures. Second, we have developed a new parallel algorithm that is multi-core

aware for RNA secondary structure prediction and optimized it for the three HPC architectures. Following this, we have performed large-scale experiments using both these algorithms on three different HPC architectures and studied their performance trends both within and across the architectures. We believe our work is the first of its kind to do a large-scale comparative analysis across three different HPC architectures in a single experimental study. Both of our contributions have been peer-reviewed by the academic community and subsequently published in leading conferences and journal.

6.1 Major Contributions

Our first major contribution is the parallelization and performance evaluation with characterization of the PKNOTS algorithm. We studied the algorithm from the design perspective and unraveled the dynamic programming recursions using the reference implementation. Following this, we developed optimized versions of the algorithm for three industry-leading HPC system architectures. Large-scale experiments using hundreds of RNA sequences were conducted on varying configurations of the HPC systems. Thousands of performance runtime data points were collected and using them we generated the algorithmic performance trends on different HPC architectures. We pushed the limits of the HPC systems to its maximum to observe the system's behavior under high workload stress.

Our second major contribution is the design and development of a new RNA

secondary structure prediction algorithm. Using our experience of parallelizing PKNOTS on various architectures, we developed this new algorithm MARSs. MARSs has been designed to exploit the explicit parallelization provided by today's multi-core architecture. It uses task-based parallelization and optimized versions of the software have been developed for both homogenous and heterogeneous multi-core architectures. The algorithm is agile by design and this property helped to move the algorithm to GAE. Again, large-scale experiments were conducted and the performance trends were studied.

This thesis has contributed the following to the parallel computing domain

1. The challenges that needs to be overcome while parallelizing an existing HPC algorithm on parallel computers
2. Performance trends of parallelized DP algorithms on homogeneous, heterogeneous and cloud-based parallel architectures
3. Performance trends of the parallel-by-design MARSs algorithm on homogeneous, heterogeneous and cloud-based parallel architectures

6.2 Future Work

Using the experience gained from this research, we suggest certain future work that can be done to both enhance and improve on our contributions. We classify them as shorter-term enhancements and longer-term improvements based on the

type & amount of work needed.

6.2.1 Short-term Enhancements

At the time of our experiments, the version of Python SDK on the Google App Engine (GAE) was 1.4.3. As of March 27 2012, Google has upgraded the Python SDK to version 1.6.4 and has added several new features. Some of the new features such as "Backend Instances" and "Pull-up Task Queues" have the potential to improve the performance of HPC applications. In addition, Google is currently adding support for RDBMS style datastore, unlimited storage and many more HPC-friendly features. As a shorter-term enhancement both MARSs and P-PKNOTS can take advantage of this newer features for better performance gains.

In our experiments on Intel x64 and IBM Cell Broadband Engine we used the generic Linux kernels that shipped with the respective Linux distributions. It would be interesting to switch the kernels with their low-latency and real-time variants and the experiments redone to observe the performance gains.

6.2.2 Long-term Improvements to MARSs Algorithm

MARSs does not currently use any heuristics in the prediction process. It is known that heuristics can speedup the evaluation of sub-structures in practice. One way to do this, is for every folding to keep track of the most stable structure of any of its subsequences using lookup tables. This can then be used as a reference to

reduce or even eliminate the evaluation of future similar sub-sequences, as it is evident that they cannot be more stable than the most stable structure closed by that base pair found so far.

MARSs currently aims to predict accurate secondary structures. As MARSs evaluates large number of alternate structures at multiple stages, it could be beneficial to switch to approximate intermediate structure predictions from a time complexity perspective.

GPU is one of the alternate HPC architectures and was beyond the scope of this thesis. It would be interesting to develop & optimize MARSs for a leading HPC architecture such as Nvidia Fermi and study the performance trends. Along the same lines it would be interesting to debate on the suitability of MARSs for the upcoming Petascale computing.

In summary, single-package multi-core architectures is the latest addition to the HPC arena and will be used more often in scientific computing fields such as bioinformatics. We plan to continue our research work in this domain for the next couple of years.

Appendices

Appendix A

Google App Engine

Google App Engine (GAE) is a scalable cloud-hosted platform built & maintained by Google and delivered as PaaS (Platform As A Service) to the software developers. GAE is built using redundant data centers located around the globe and the developer sees it as one global platform. The application developer is provided with a secure sand-boxed & optimized language runtime. Hence it is easy to build, maintain and scale the applications. Everything under the runtime, namely the operating system (OS), software, patches, backup, hardware, networking is maintained by Google transparently.

From a GAE-hosted application‘s perspective, the various system components like CPU, Memcache, and non-volatile storage are exposed as services through well-defined APIs (Application Programming Interfaces). Any application that needs these services should request them by invoking the respective APIs. By this way,

GAE makes it very easy to build an application that runs reliably, even under heavy load and with large amounts of data. Google App Engine includes the following features

- Dynamic web serving, with full support for common web technologies
- Persistent storage with queries, sorting and transactions
- Automatic scaling and load balancing
- APIs for authenticating users and sending email using Google Accounts
- A fully featured local development environment that simulates Google App Engine
- Task queues for performing work outside of the scope of a web request
- Scheduled tasks for triggering events at specified times and regular intervals

GAE applications can run in one of three runtime environments: the 'Go' environment, the Java environment, and the Python environment. Each environment provides standard protocols and common technologies for web application development. Our application is developed in the Python programming language and hence uses the python environment. The following are some of the major features of Google App Engine that we use in our application.

The Sandbox Applications run in a secure environment that provides limited access to the underlying operating system. The sandbox isolates the ap-

plication in its own secure, reliable environment that is independent of the hardware, operating system and physical location of the web server. These limitations allow App Engine to distribute web requests for the application across multiple servers, and start & stop the servers to meet traffic demands.

Python Runtime The Python runtime environment uses Python version 2.5.2. It supports the Python standard library except for a few features that will defeat the sandbox like opening direct network connections to other computers, sockets to non-standard ports. In addition, the python environment provides rich Python APIs for the datastore, Google Accounts, URL fetch, and email services. App Engine also provides a simple Python web application framework called webapp to make it easy to start building applications. The compiled python application code is cached for rapid responses to web requests. External python libraries can be uploaded as long as they do not violate the sandbox or require the unsupported python standard libraries. Python extensions written in 'C' language are not supported.

Datastore Google App Engine provides a distributed data storage service that features a query engine and transactions. The Appengine datastore is unlike a traditional relational database. Data objects, or "entities", have a kind and a set of properties. Queries can retrieve entities of a given kind filtered and sorted by the values of the properties. Several types of property values are supported. Datastore entities are "schema-less". The structure of data entities is provided by and enforced by the application code. The datastore

is strongly consistent and uses optimistic concurrency control. The datastore can be used with a SQL (Structured Query Language) like query language called GQL (Google Query Language). Certain features of SQL such as 'Join', 'LIKE', 'Not equal', 'OR' are not supported by the datastore though.

Memcache The Memcache service provides the application with a high performance in-memory key-value cache that is accessible by multiple instances of the application. Memcache is useful for data that does not need the persistence and transactional features of the datastore, such as temporary data or data copied from the datastore to the cache for high-speed access.

Instances Appengine applications are web application by nature and therefore execute in response to a web request from a client over the http/s protocol. When a new request is received, the Appengine will either spin a new instance or recycle an existing one. The number of concurrent instances at any one time is determined by a scheduler that consider the rate at which web requests are received, serviced and the application latency before a response is sent to the client. The start-up time for an instance can be reduced by using warm-up requests and also by using instance of type "always on".

Task Queues An application can also perform work outside of responding to a web request, using tasks. Tasks are small, discrete-units of code that are scheduled using task queue APIs. Tasks are similar to 'threads' in POSIX standard except that they do not share a single global namespace. The application can perform tasks on a schedule that is configurable, such as

on a daily or hourly basis. Alternatively, the application can perform tasks added to a queue by the application itself, such as a background task created while handling a request. Background tasks created using the task queue API can only run up to a maximum of 10 minutes. After this, the Appengine scheduler will kill the task. In order to continue executing, the task needs to queue another task and restart from where it left off. This is called as task chaining. Recently, Google added a new type of background instance called Backends that has no execution time limit. Backends are not covered here as we have not used this feature in our software development.

System constraints The table A.1 lists the most common limitations of various GAE system components at the time our experiments were conducted. The SDK (Software Development Kit) version used was 1.4.3. These limitations have direct impact on how an application can be structured in-order to run it on GAE.

Table A.1: GAE System Constraints

HTTP/S request handlers	• Maximum 30 seconds runtime • Request size - 10MB • Response size - 10MB
Default Instances	• Maximum memory size 300MB • CPU speed (variable), up to 1.2 GHz • Run in separate namespace
Datastore	• Entity size limited to 1MB • Higher latency compared to Memcache • Datastore contention during simultaneous writes
Memcache	• Object size limited to 1MB • Number of objects limited by (undisclosed) capacity • Objects may be deleted at anytime
Background Tasks	• 10 task queues for free apps • Task object size limited to 10KB • Maximum runtime of 10 minutes

Appendix B

Intel x64

A multi-core processor is a single computing component with two or more independent actual processors (called "cores"), which are the units that read and execute software instructions. In essence, a multi-core processor features multiple CPUs in a single physical package. A multi-core processor may or may not share a single cache.

Intel x64 is a 64-bit architecture and is an extension to the popular 32-bit x86 architecture and was originally invented by AMD and called AMD64. This architecture allows the programs to easily refer to much larger virtual and physical address space. Because the full 32-bit architecture remains implemented in the hardware without any intervening emulation, existing 32-bit executable programs will run with no compatibility and performance penalties.

The primary defining characteristic of AMD64 is the availability of 64-bit general-

Table B.1: Intel System Specifications

Chip Architecture	Intel x64
CPU Model	E7450
CPU Clock speed	2.4Ghz
FSB Speed	1066MHz
No. Of Physical CPUs	2
No. Of Cores per CPU	6
Level '3' Cache Size	12 MB
RAM Size	64 GB

purpose processor registers, 64-bit integer arithmetic & logical operations, and 64-bit virtual address space. In addition, the number of named general-purpose registers is increased from eight (i.e. eax, ebx, ecx, edx, ebp, esp, esi, edi) in x86 to 16 (i.e. rax, rbx, rcx, rdx, rbp, rsp, rsi, rdi, r8, r9, r10, r11, r12, r13, r14, r15) in x64. It is therefore possible to keep more local variables in registers rather than on the stack, and to let registers hold frequently accessed constants; arguments for small and fast subroutines may also be passed in registers to a greater extent. However, AMD64 still has fewer registers than RISC-based processors like CBEA, which has 128 registers.

The machine used in this project consists of two physical processors of Intel®Xeon®Processor E7450 with six independent cores each running at 2.4 GHz. Table B.1 shows the major specifications of the processor.

Appendix C

IBM Cell Broadband Engine

Cell is a heterogeneous multi-core architecture and is the shorthand for Cell Broadband Engine Architecture, commonly abbreviated as either CBEA or Cell BE. Cell combines general-purpose power architecture of modest performance with streamlined co-processing elements that greatly accelerates multimedia and vector processing applications, as well as many other forms of dedicated computations.

The Cell processor can be split into four components: external input and output structures, the main processor called the Power Processing Element (PPE) - a two-way simultaneous multi-threaded Power ISA v.2.03 compliant core, eight fully functional co-processors called the Synergistic Processing Elements, or SPEs, and a specialized high-bandwidth circular data bus connecting the PPE, input/output elements and the SPEs, called the Element Interconnect Bus or EIB.

PPE The PPE is Power Architecture based two-way multi-threaded core. The

PPE contains a 64 KB level 1 cache (32 KB instruction and a 32 KB data) and a 512 KB Level 2 Cache. PPE is capable of producing 6.4 GFLOPS double-precision calculations or 25.6 GFLOPS of single-precisions calculations at 3.2 GHz. However, PPE is usually used to run conventional operating systems and as a manager of SPEs.

SPE The Cell processor has 8 SPEs. An SPE is a RISC processor with 128-bit SIMD organization for single and double precision instructions. The SPEs contain a 128-bit, 128-entry register file. Each SPE is composed of a Synergistic Processing Unit (SPU) and a Memory Flow Controller (MFC). Each SPE contains a 256 KB SRAM for instruction and data called "Local Storage". The SPU cannot directly access the main memory; MFC has to set up a DMA operation to retrieve the data from the main memory address space to its local storage. At 3.2 GHz, each SPE gives a theoretical 25.6 GFLOPS of single precision performance. For double-precision floating-point operations, Cell performance drops by an order of magnitude, but still reaches 20.8 GFLOPS (1.8 GFLOPS per SPE, 6.4 GFLOPS per PPE). The PowerXCell 8i variant, which was specifically designed for double precision, reaches 102.4 GFLOPS in double-precision calculations.

EIB The EIB is a communication bus internal to the Cell processor that connects the various on-chip system elements: the PPE, the memory controller (MIC), the eight SPE coprocessors, and two off-chip I/O interfaces. The EIB is implemented as a circular ring comprising of four 16B-wide unidirectional

channels, which counter-rotate in pairs. EIB runs at half the system clock rate and hence the effective channel rate is 16 bytes every two system clocks. At maximum concurrency, the peak instantaneous EIB bandwidth is 96B per clock (12 concurrent transactions * 16 bytes wide / 2 system clocks per transfer).

Peripherals Cell contains a dual channel Rambus XIO macro, which interfaces to Rambus XDR memory. The memory interface controller (MIC) is separate from the XIO macro. The XIO-XDR link runs at 3.2 Gbps per pin. Two 32-bit channels can provide a theoretical maximum of 25.6 GB/s. The I/O interface, also a Rambus design, is known as FlexIO. This provides a theoretical peak bandwidth of 62.4 GB/s (36.4 GB/s outbound, 26 GB/s inbound) at 2.6 GHz.

Limitations The Cell architecture emphasizes efficiency/watt, prioritizes bandwidth over latency, and favors peak computational throughput over simplicity of program code. For these reasons, Cell is widely regarded as a challenging environment for software development.

Table C.1 shows the system specifications of the Cell blade server and the PS3. Figure C.1 shows the Cell Processor Schematic. The Cell blade server has 8 SPEs while the Sony PS3 has only 6 SPEs.

Table C.1: Cell System Specifications

PPE Architecture	Power Architecture based dual-threaded core
PPU CPU Clock Speed	GHz
Number of SPEs	8 in PowerXcell 8i, 6 in Sony PS3
PPE CPU Clock Speed	GHz
PPE Local Storage Size	256KB

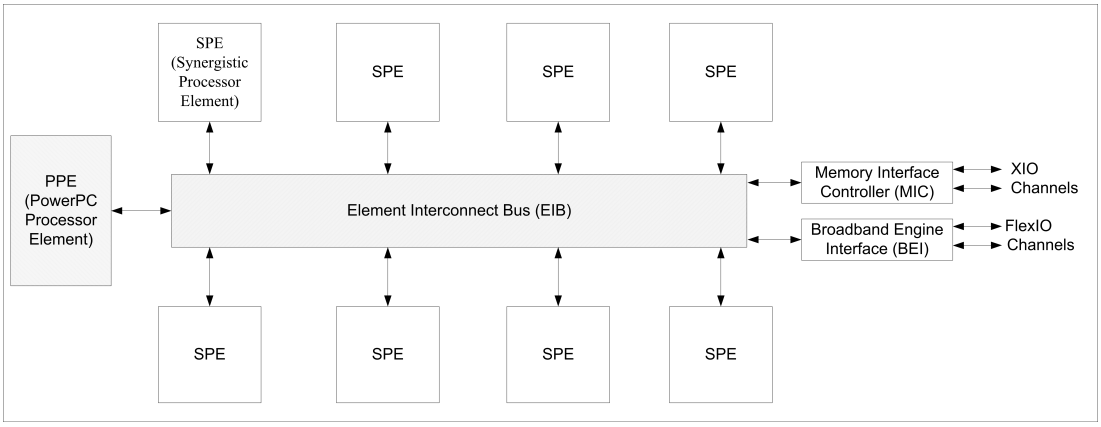


Figure C.1: Cell Microprocessor Schematic

Appendix D

A Brief History of Early Parallel Computing Architectures

Industry & Academia have invented and evolved many different types of parallel computing architectures - SMPs, Cluster Computing, Grid Computing and more recently Multi-Cores. Each of these architectures have their strengths & weaknesses and pose unique challenges in software development.

D.1 Symmetric Multi-Processing

SMPs were the first to arrive in the scene of parallel computing space. SMP is a computer hardware architecture in which two or more identical processors are connected to the single shared main memory and are controlled by a single OS

instance. SMP systems allow any processor to work on any task no matter where the data is in the memory. This type of architecture where the data is equidistant to all the processors is also known as uniform memory access.

The strength of this architecture is the ease of programmability as any given task instance still runs on only one processor in the system at any given time. More than one instance of the same task processing different data ranges can be executed concurrently (up to the maximum number of CPUs) and simultaneously beyond that with performance penalty. The weakness of the architecture is the scalability. This affects both CPU-bound and IO-bound processes. Running significantly higher number of CPU-bound processes than the available processors results in performance degradation of all the processes as the OS needs to save/resume the individual processes. For IO-bound processes, the memory bandwidth becomes the bottleneck as processes compete amongst themselves to read/write to the main memory. Ultimately, the memory bandwidth places the upper bound on the number of processors in a SMP system. This situation is made more complicated by memory wall and frequency wall. Figure D.1 shows a schematic of a typical SMP system.

D.2 Cluster Computing

Following SMPs, the next parallel computing architecture to be widely researched and deployed is the cluster computing architecture. A computing cluster seeks

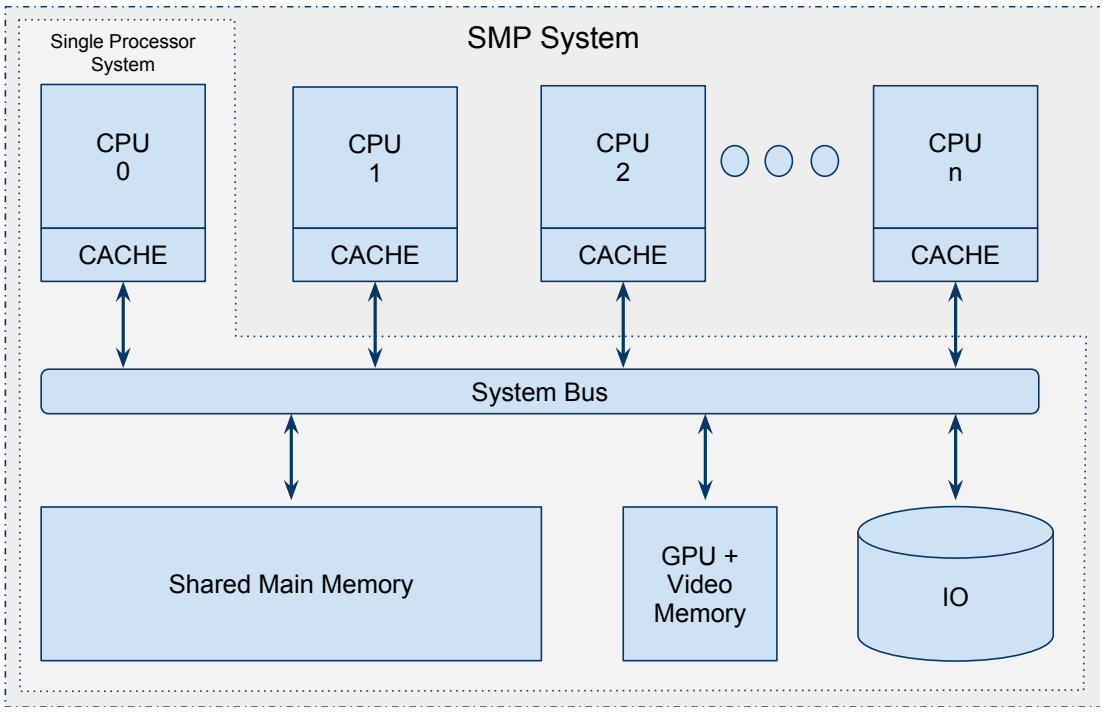


Figure D.1: Symmetric Multiprocessing Schematic

to make it possible to group together arbitrary number of processors as a single computing system. The system architecture links multiple independent computing nodes over a high-speed network backend. Each of the computing nodes can be single-CPU or a SMP system running an instance of the same operating system. One of the nodes serve as the head that distribute workloads to the rest of the computing nodes known as the worker nodes. A computer cluster is usually built from scratch and when assembled from COTS (Common Off The Shelf) components is known as the Beowulf cluster.

The strength of this architecture is the ability to add arbitrary number of processors and distribute tasks among them. There are several weakness in this architecture. First, the programming environment is challenging as the application design

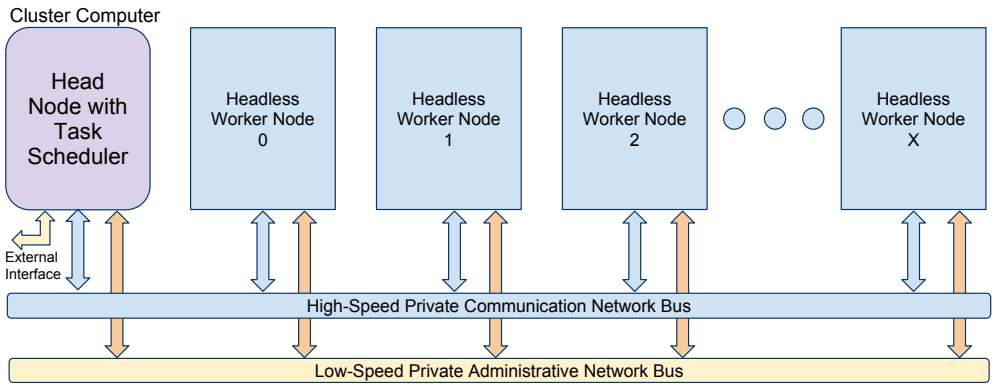


Figure D.2: Cluster Computing Schematic

needs to accommodate the communication requirements (and the associated communication delay) between tasks running in multiple nodes. Second, unlike a SMP where each processor has access to the entire main memory, a node in a computing cluster is generally limited to the main memory within itself. Although it is possible to use remote main memories as extension to local main memory, practical considerations like node-failure, data redundancy & migration, memory access-latencies, cache-invalidation limits the practical usefulness of the same. Figure D.2 shows a schematic of a typical computer clustered system.

D.3 Grid Computing

Closely related to cluster computing is grid computing. In the case of cluster computing, the various computing nodes are tightly coupled and communicate over dedicated high-speed communication channel and usually are composed of homogenous nodes. A cluster computer is therefore suitable for HPC jobs. In

contrast, a grid is an amalgamation of computing resources over a relatively slow communication channel like internet. Grids are loosely coupled, heterogeneous and widely dispersed. Therefore a grid is suited for jobs that run independently of each other and communicate their results to a coordinating server.

The strength of the grid architecture is that the size can vary by a considerable amount and within a short span of time. For example, SETI@Home project claims to be the largest distributed computing project with over 3 million active users or nodes. The major weakness of the grid computing is the non-uniformity of the hardware & software on the computing nodes. Factors like CPU type, bus speed, RAM size, network bandwidth & latency are highly variable limiting the performance and size of the task that can be run on the nodes. On the software side, each node is in a different administrative domain and may run different OS, version as well. These factors make the grid computing unsuitable for applications of HPC class. Figure D.3 shows a schematic of typical grid architecture.

D.4 Multi-core Computing

Multi-core processors is the latest HPC architecture to be invented. A multi-core processor is a single computing component that has two or more actual processors (or "cores") integrated into a single integrated circuit die or onto multiple dies in a single chip package.

Processors were originally designed to be single-cores. Two or more such processors

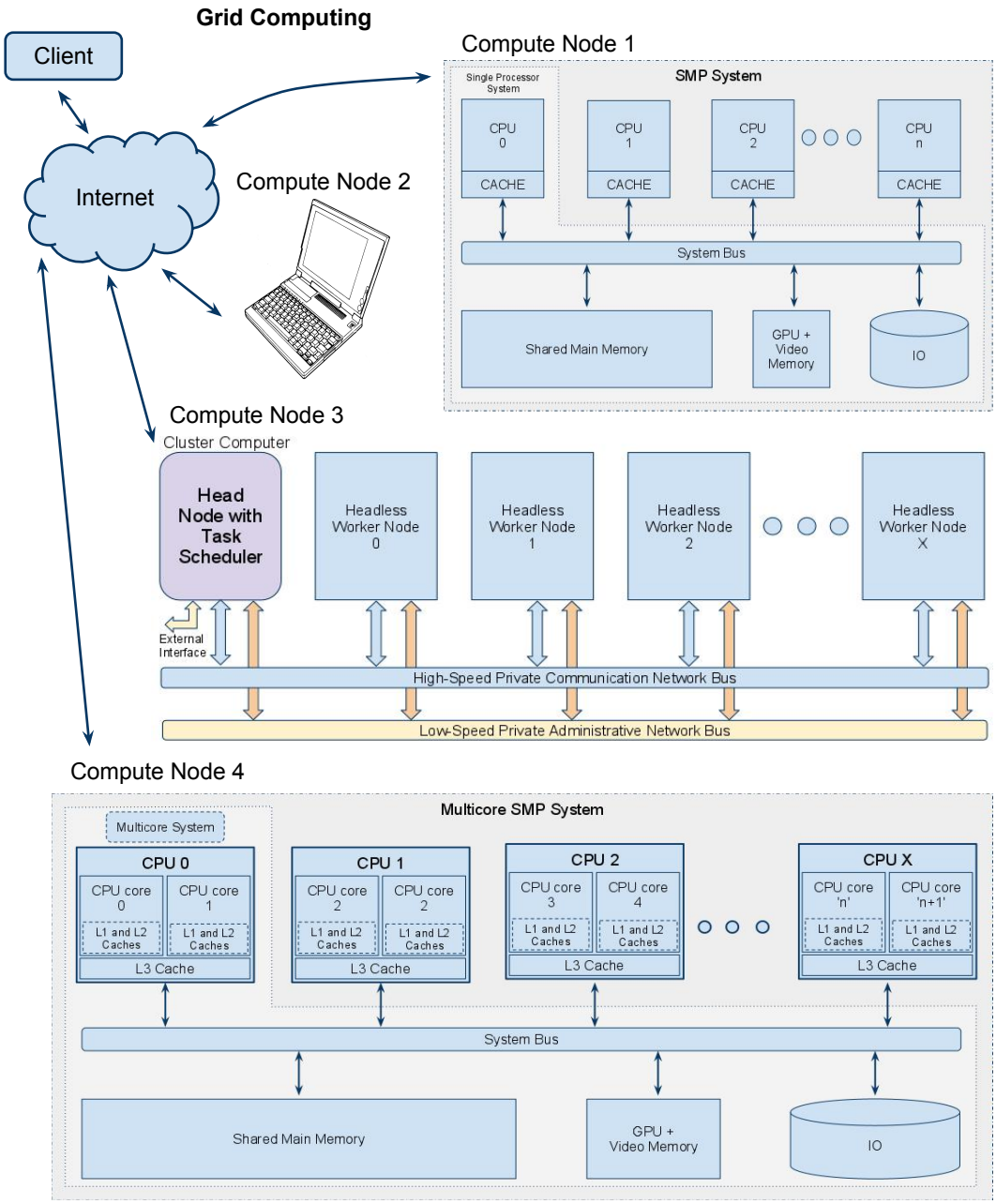


Figure D.3: Grid Computing Schematic

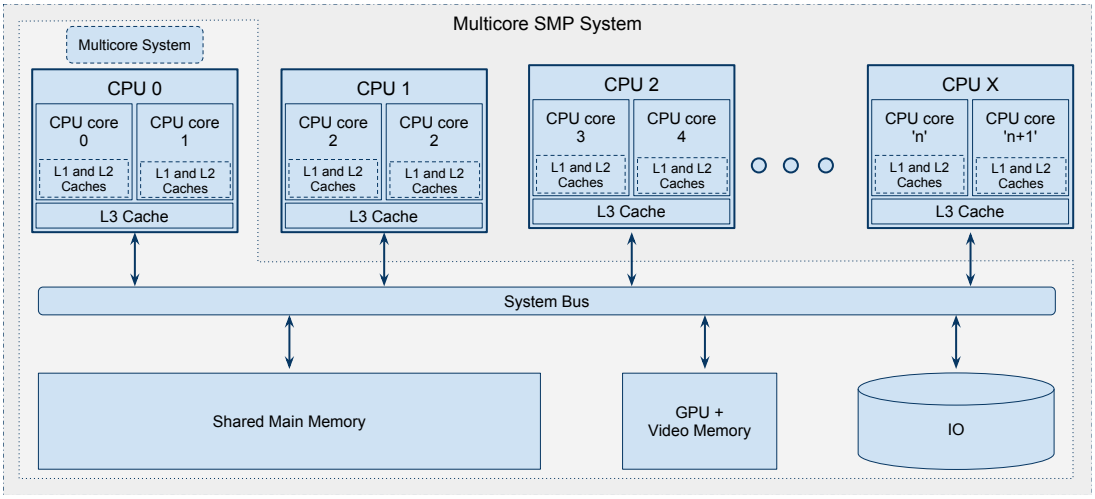


Figure D.4: Multicore Computing Schematic

were combined to form a SMP and as highlighted in a previous section, this architecture has its limitations due to the limited bandwidth of the shared memory bus. In a multi-core CPU two or more cores can communicate directly thereby taking the load of the shared system bus. Multi-cores can be either loosely or tightly coupled depending on if they share a common cache or not. Also, multi-cores are classified into homogenous and heterogeneous. In a homogenous multi-core all the cores are of the same type and implement the same Instruction Set Architecture (ISA). In a heterogeneous architecture, the cores can be of several types and implement different ISAs. Example of homogeneous multi-core processors are Intel Core 2 Duo while IBM Cell is a heterogeneous multi-core processor implementing two different ISAs. Figure D.4 shows the schematics of both a single multi-core and a multi-core SMP system.

HPC tasks are characterized as needing large amounts of computing power over short periods of time. This can be achieved by performance scaling of a single

processor or parallel scaling using multiple processors. It has been explained in an earlier section that performance scaling of a single processor has reached its practical limits and recently the industry is achieving higher performance using parallel scaling. Among the three parallel architectures discussed, multi-core systems are the latest and least investigated. Therefore, this thesis examines the performance characteristics of two multi-core architectures, one being homogenous type and other heterogeneous type.

Bibliography

[1] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities", *AFIPS Conference Proceedings* , Volume 30, April 1967.

[2] J. P. Abrahams, M. van den Berg, E. van Batenburg, C. Pleij, "Prediction of RNA secondary structure, including pseudoknotting, by computer simulation", *Nucleic Acids Research*, Volume 18, Issue 10, 25 May 1990, Pages 3035-3044.

[3] V. R. Akmaev, S. T. Kelley, and G. D. Stormo, "A phylogenetic approach to RNA structure prediction", *Proc Int. Conf. Intell. Syst. Mol. Bio.*, 1999, Pages 10-7.

[4] T. Akutsu, "Dynamic programming algorithms for RNA secondary structure prediction with pseudoknots", *Discrete Applied. Mathematics - Special volume on combinatorial molecular biology*, Volume 104, Issue 1-3 , 15 August 2000 , Pages 45-62.

[5] J. Anvik, S. MacDonald, D. Szafron, J. Schaeffer, S. Bromling, Kai Tan, "Generating Parallel Programs from the Wavefront Design Pattern", *Parallel and Distributed Processing Symposium* , 07 August 2002, Pages 104 - 111.

[6] C. E. R. Alves, E. N. CÃąceres, F. Dehne, S. W. Song, "A Parallel Wavefront Algorithm for Efficient Biological Sequence Comparison", *Computational Science and Its Applications* , 2003, Pages 249-258.

[7] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, K. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley", *EECS Department University of California, Berkeley, Technical Report*, 18 December 2006.

[8] F. H. D. van Batenburg, A. P. Gultyaev, C. W. A. Pleij, "An APL-programmed genetic algorithm for the prediction of RNA secondary structure", *Journal of Theoretical Biology*, Volume 174, Issue 3, 7 June 1995, Pages 269-280.

[9] M. Brown, C. Wilson, "RNA Pseudoknot Modeling Using Intersections of Stochastic Context Free Grammars with Applications to Database Search", *Pacific Symposium on Biocomputing*, 1996 , Pages 109-125.

[10] H. M. Berman, A. Gelbin, J. Westbrook, L. Clowney, C. Zardecki, "The Nucleic Acid Database: Present and future", *Journal of Research of the National Institute of Standards and Technology*, Volume 101, Issue 3, May 1996, Pages 243.

[11] H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. N. Bhat, H. Weissig, I. N. Shindyalov, P. E. Bourne, "The Protein Data Bank", *Nucleic Aicds Research* , Volume 28, Issue 1, 17 October 1999, Pages 235 - 242.

[12] F. H. D. van Batenburg, A. P. Gultyaev, C. W. A. Pleij, "PseudoBase: structural information on RNA pseudoknots", *Nucleic Acids Research*, Volume 29, Issue 1, 1 January 2001, Pages 194-195.

[13] D.A. Bader, V. Agarwal, K. Madduri, "On the design and analysis of irregular algorithms on the Cell processor: A case study of list ranking", *In Proceedings 21st IEEE International. Parallel and Distributed Processing Symposium* , March 2007, Pages 1-10.

[14] S. Bellaousov, D. H. Mathews, "ProbKnot: Fast prediction of RNA secondary structure including pseudoknots", *RNA Journal*, Volume 16, Issue 10, 16 October 2010, Pages 1870-1880.

[15] M. Bon, H Orland, "TT2NE: a novel algorithm to predict RNA secondary structures with pseudoknots", *Nucleic Acids Research* 18 May 2011, Pages 1-8.

[16] R. B. Cary, G. D. Stormo, "Graph-Theoretic Approach to RNA Modeling Using Comparative Data", *Proceedings International Conference on Intelligent Systems for Molecular Biology 1995* , Volume 3, Pages 75-80.

[17] J. J. Cannone, S. Subramanian, M. N. Schnare , J. R. Collett, L. M. D'Souza, Y. Du, B Feng, N. Lin, L. V. Madabusi, K. M. M§ller, N. Pande, Z. Shang,

N. Yu, R. R. Gutell, "The Comparative RNA Web (CRW) Site: An Online Database of Comparative Sequence and Structure Information for Ribosomal, Intron, and Other RNAs.", *BMC Bioinformatics*, Volume 3, Issue 2, 17 January 2002.

[18] L. Chai, Q. Gao, D. K. Panda, "Understanding the Impact of Multi-Core Architecture in Cluster Computing:Case Study with Intel Dual-Core System", *International Sympsoium on Cluster Computing and the Grid*, 29 May 2007, Pages 417-478.

[19] Xiaowen Chen, Zhonghai Lu, A. Jantsch, Shuming Chen, "Speedup Analysis of Data-parallel Applications on Multi-core NoCs", *ASIC IEEE 8th International Conference*, 11 December 2009, Pages 105-108.

[20] A. Condon, H. Jabbari, "Computational prediction of nucleic acid secondary structure: Methods, applications, and challenges", *Theoretical Computer Science*, Volume 410, Issue 4-5, 17 February 2009, Pages 294-301.

[21] E. Dam, K. Pleij, D. Draper, "Structural and functional aspects of RNA pseudoknots", *Biochemistry*, Volume 31, Issue 47, 1 December 1992, Pages 11665- 11676.

[22] J. A. Doudna, T. R. Cech, "The chemical repertoire of natural ribozymes", *Nature*, Volume 418, Issue 6894, 11 July 2002, Pages 222-228.

[23] Ye Ding,C. E. Lawrence, "A statistical sampling algorithm for RNA secondary structure prediction", *Nucleic Acids Research*, Volume 31 , Issue 24, 29 October 2003, Pages 7280-7301.

[24] J. S. Deogun, R. Donts, O. Komina, Fangrui Ma, "RNA Secondary Structure Prediction with Simple Pseudoknots", *Asia-Pacific Bioinformatics Conference - APBC*, Volume 29, January 2004, Pages 239-246.

[25] R. D. Dowell, S. R. Eddy, "Evaluation of several lightweight stochastic context-free grammars for RNA secondary structure prediction", *BMC Bioinformatics*, Volume 5, Issue 1, 4 June 2004, Pages 5-71.

[26] A. Deschenes, K. C. Wiese, J. Poonian, "Comparison of dynamic programming and evolutionary algorithms for RNA secondary structure prediction", *Computational Intelligence in Bioinformatics and Computational Biology*, October 2004, Pages 214-222.

[27] Y. DING, C. Y. CHAN, C. E. LAWRENCE, "RNA secondary structure prediction by centroids in a Boltzmann weighted ensemble", *RNA* , Volume 11, Issue 8, 2005, Pages 1157-1166.

[28] C. B. Do, D. A. Woods, S. Batzoglou "CONTRAfold: RNA secondary structure prediction without physics-based models", *Bioinformatics* , Volume 22, Issue 14, 2006, Pages e90-e98.

[29] J. Edmonds, "Maximum matching and polyhedron with 0, 1-vertices", *J. of Research National. Bureau of Standards Section B* , 1965, Pages 125-130.

[30] S. R. Eddy, "Noncoding RNA genes and the modern RNA world", *Nature Reviews Genetics*, Volume 2, Issue 12, December 2001, Pages 919-929.

[31] S. R. Eddy, "How do RNA folding algorithms work?", *Nature Biotechnology* , Volume 22, Issue 11, 2004, Pages 1457-1458.

[32] T. Estrada, A. Licon, M. Taufer, "compPknots - a Framework for Parallel Prediction and Comparison of RNA Secondary Structures with Pseudoknots", *Frontiers of High Performance Computing and NetworkingâĂŤISPA 2006 Workshops*, 2006, Pages 677-686.

[33] S. M. Freier, R. Kierzek, J. A. Jaeger, N. Sugimoto, M. H. Caruthers, T. Neilson, D. H. Turner, "Improved free-energy parameters for predictions of RNA duplex stability", *Proceedings of the National Academy of Sciences of the United States of America*, Volume 83, Issue 24, December 1986, Pages 9373-9377.

[34] M. Fekete "Prediction of RNA Secondary Structures using Parallel Computers", *Thesis* , 1997, Pages 10âĂŞ14.

[35] H. N. Gabow, "An Efficient Implementation of Edmonds' Algorithm for Maximum Matching on Graphs", *Journal of the ACM*, Volume 23, Issue 2, April 1976, Pages 221-234.

[36] R. R. Gutell, A. Power, G. Z. Hertz, E. J. Putz, G. D. Stormo, "Identifying constraints on the higher-order structure of RNA: continued development

and application of comparative sequence analysis methods", *Nucleic Acids Research*, Volume 20, Issue 21, 11 November 1992, Pages 5785-5795.

[37] A. P. Gultyaev, F. H. van Batenburg, C. W. Pleij, "The computer simulation of RNA folding pathways using a genetic algorithm", *Journal of Molecular Biology* , Volume 250, Issue 1, 30 June 1995, Pages 37-51.

[38] J. Gorodkin, L. J. Heyer, G. D. Stormo, "Finding the most significant common sequence and structure motifs in a set of RNA sequences", *Nucleic Acids Research*, Volume 25, Issue 28, 15 September 1997 ,Pages 3724-3732.

[39] A. P. Gultyaev, F. H. van Batenburg, C. W. Pleij, "An approximation of loop free energy values of RNA H-pseudoknots", *RNA*, Volume 5, Issue 4, 8 September 2000, Pages 609-617.

[40] J. Gorodkin, S. L. Stricklin, G. D. Stormo, "Discovering common stem-loop motifs in unaligned RNA sequences", *Nucleic Acids Research*, Volume 29, Issue 10, 15 May 2001 , Pages 2135-2144.

[41] M. L. Green, R. Miller, "Molecular structure determination on a computational and data Grid", *Journal Parallel Computing - Special issue: High-performance parallel bio-computing* , Volume 30, Issue 9-10, 27 September 2004, Pages 1001-1017.

[42] I. L. Hofacker, W. Fontana, P. F. Stadler, L. S. Bonhoeffer, M. Tacker, P. Schuster, "Fast Folding and Comparison of RNA Secondary Structures",

Monatshefte for Chemie Chemical Monthly , Volume 125, Issue 2, 1994, Pages 167 - 188.

[43] K. Han, D. Kim, H. J. Kim, "A Vector-based Method for drawing RNA Secondary Structure", *Bioinformatics*, Volume 15, Issue 4, 1999, Pages 286-297.

[44] C. Haslinger, "Prediction Algorithms for Restricted RNA Pseudoknots", *PhD thesis*, Universitat Wien, March 2001.

[45] T. Hoefler, "The Cell Processor - A short Introduction", *22 Chaos Communication Congress*, 12 December 2005, Pages 286-292.

[46] M. D. Hill, M. R. Marty, "Amdahls Law in the Multicore Era", *Computer*, Volume 41, Issue 7, 15 July 2008, Pages 33-38.

[47] M. Hamada, H. Kiryu, K. Sato, T. Mituyama, K. Asai, "Prediction of RNA secondary structure using generalized centroid estimators", *Bioinformatics* , Volume 25, Issue 4, 2009, Pages 465-473.

[48] J. A. Jaeger, Jr. J. SantaLucia, I. Jr. Tinoco, "Determination of RNA structure and thermodynamics", *Annual Review of Biochemistry* , Volume 62, 1993, Pages 255-287.

[49] J. Kim, J. R. Cole, S. Pramanik, "Alignment of possible secondary structures in multiple RNA sequences using simulated annealing", *Computer Applications in Biosciences*, Volume 12, Issue 4, August 1996, Pages 259-267.

[50] M. H. Kolk, M.van der Graff, S. S. Wijmenga, C. W. Pleij, H. A. Heus, C. W. Hilbers, "NMR structure of a classical pseudoknot: interplay of single- and double-stranded RNA", *Science* , Volume 280, Issue 5362, 17 April 1998, Pages 434-438.

[51] H. Karen, "When RNA ruled another lost world?", *HMS Beagle The BioMed-Net Magazine*, 27 March 1998.

[52] S. Kawaharaa, T. Uchimaru, M. Sekine, "The hydrogen bond energy on mismatched base pair formation between uracil derivatives and guanine in the gas phase and in the aqueous phase", *Journal of Molecular Structure: THEOCHEM*, Volume 530, Issue 1-2, 18 September 2000, Pages 109-117.

[53] S. Kawahara, T. Uchimaru, "Computer-Aided Molecular Design of Hydrogen Bond Equivalents of Nucleobases: Theoretical Study of Substituent Effects on the Hydrogen Bond Energies of Nucleobase Pairs", *European Journal of Organic Chemistry*, Volume 2003, Issue 14,26 June 2003, Pages 2577âĂŞ2584.

[54] B. Knudsen, J. Hein "Pfold: RNA secondary structure prediction using stochastic context-free grammars", *Nucleic Acids Research* , Volume 31, Issue 13, 2003, Pages 3423-3428.

[55] P. Kongetira, K. Aingaran, K. Olokotun, " Niagara: A 32-Way Multithreaded Sparc Processor", *IEEE Micro*, Volume 25, Issue 2, March 2005, Pages 21-29.

[56] R. Kota, R. Oehler, " Horus: Large-Scale Symmetric Multi-processing for Opteron Systems", *IEEE Micro*, Volume 25, Issue 2, March 2005, Pages 30-40.

[57] RB Lyngso, M Zuker,C. N. S. Pedersen, "An Improved Algorithm for RNA Secondary Structure Prediction", *BRICS Research Series*, May 1999, Page 24, RS-99-15.

[58] R. B. Lyngso, C. N. S. Pedersen, "Pseudoknots in RNA secondary structures", *RECOMB00: Proceedings of the Fourth Annual International Conference on Computational Molecular Biology* , 2000, Pages 201-209.

[59] T. Liu, B. Schmidt, "Parallel RNA secondary structure prediction using stochastic context-free grammars", *Journal Concurrency and Computation: Practice & Experience* ,Volume 17, Issue 14, 10 December 2005, Pages 1669âĂŞ1685.

[60] J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, J. Dongarra, "Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems)", *Proceedings of the ACM/IEEE conference on Supercomputing* , 2006, Pages 113.

[61] Z. J. Lu, J. W. Gloor, D. H. Mathews, "Improved RNA secondary structure prediction by maximizing expected pair accuracy", *RNA*, Volume 15, Issue 10, October 2009, Pages 1805-1813.

[62] J. S. McCaskill, "The equilibrium partition function and base pair binding probabilities for RNA secondary structure", *Biopolymers*, Volume 29, Issue 6-7, May-June 1990, Pages 1105-1119.

[63] W. S. Martins , J. B. Del Cuvillo, F. J. Useche, K. B. Theobald, G.R. Gao, "A multithreaded parallel implementation of a dynamic programming algorithm for sequence comparison", *Pacific Symposium on Biocomputing* , 2001, Pages 311-322.

[64] D. H. Mathews, M. D. Disney, J. L. Childs, S. J. Schroeder, M. Zuker, D. H. Turner, "Incorporating chemical modification constraints into a dynamic programming algorithm for prediction of RNA secondary structure", *Proceedings of the National Acadamey of Sciences USA*, Volume 101, Issue 9, 11 May 2004, Pages 7287-7292.

[65] C. McNairy, R. Bhatia, " Montecito: A Dual-Core, Dual-Thread Itanium Processor", *IEEE Micro*, Volume 25, Issue 2, March 2005, Pages 10-20.

[66] R. Nussinov, A. B. Jacobson, "Fast algorithm for predicting the secondary structure of single-stranded RNA", *Proceedings of the National Acadamey of Sciences USA*, Volume 77, Issue 11, November 1980 Pages 6309-6313.

[67] A. Nakaya, K. Yamamoto, A. Yonezawa, "RNA secondary structure prediction using highly parallel computers", *Computer Applications in the Biosciences* , Volume 11, Issue 6, 6 Spetmeber 1995, Pages 685-692.

[68] A. Nakaya, K. Taura, K. Yamamoto, A. Yonezawa, "Visualization of RNA secondary structures using highly parallel computers", *Bionformatics* Volume 12, Issue 3, 3 June 1996, Pages 205-211.

[69] P. Nissen, J. Hansen, N. Ban, P. B. Moore, T.A. Steitz, "The structural basis of ribosomal activity in peptide bond synthesis", *Science*, Volume 289, Issue 5481, 11 August 2000, Pages 920-930.

[70] C. W. Pleij, K. Rietveld, L. Bosch, "A new principle of RNA folding based on pseudoknotting", *Nucleic Acids Research*, Volume 13, Issue 5, 11 March 1985, Pages 1717-1731.

[71] A. Perez, J. Sponer, P. Jurecka, P. Hobza , F. J. Luque, M. Orozco, "Are the Hydrogen Bonds of RNA (A-U) Stronger Than those of DNA (A-T)? A Quantum Mechanics Study", *Chemistry*, Volume 11, Issue 17, 19 August 2005, Pages 5062-5066.

[72] J. M. Paul, B. H. Meyer, "Amdahl's Law Revisited for Single Chip Systems", *International Journal of Parallel Programming*, Volume 35, Issue 2, April 2007, Pages 101-123.

[73] F. Petrini, G. Fossum, J. Fernandez, A. L. Varbanescu, N. Kistler, M. Perrone, "Multicore Surprises: Lessons Learned from Optimizing Sweep3D on the Cell Broadband Engine", *Parallel and Distributed Processing Symposium* , 11 June 2007, Pages 1-10.

[74] B. H. Park; M. Schmidt, K. Thomas, T. Karpinets, N. F. Samatova, "Parallel, Scalable, Memory-Efficient Backtracking for Combinatorial Modeling of Large-Scale Biological Systems", *Parallel and Distributed Processing, IEEE International Symposium* , 03 June 2008, Pages 1-8.

[75] J. M. Rabaey, "Digital integrated circuits: a design perspective", *Prentice-Hall, Inc*, 1996.

[76] E. Rivas and S. Eddy, "A dynamic programming algorithm for RNA structure prediction including pseudoknots", *Journal of Molecular Biology*, Volume 285, Issue 5, 5 February 1999, Pages 2053-2068.

[77] D. Sankoff, "Simultaneous solution of the RNA folding, alignment and protosequence problems", *Journal on Applied Mathematics*, Volume 45, Issue 5, 1985, Pages 810-825.

[78] M. J. Serra, D. H. Turner, "Predicting the thermodynamic properties of RNA", *Methods Enzymol*, Volume 259, 1995, Pages 242-261.

[79] B. A. Shapiro, J. C. Wu, "Predicting RNA H-Type pseudo-knots with the massively parallel genetic algorithm", *Computer Applications in the Biosciences* , Volume 13, Issue 4 , 17 March 1997, Pages 459-471.

[80] S. J. Schroeder, M. E. Burkard ME, D. H. Turner, "The Energetics of Small Internal Loops in RNA", *Biopolymers*, Volume 52, Issue 4, 29 March 2001, Pages 157-167.

[81] B. A. Shapiro, J. C. Wu, D. Bengali, M. J. Potts, "The Massively parallel genetic algorithm for RNA folding - MIMD implementation and population variation", *Bioinformatics*, Volume 17, Issue 2, February 2001, Pages 137-148.

[82] J. Sadecki, "Parallel dynamic programming algorithms: Multitransputer systems", *Journal of applied mathematics and computer science*, Volume 12, Issue 2, 2002, Pages 241-255.

[83] M. Sprinzl , K. S. Vassilenko, "Compilation of tRNA sequences and sequences of tRNA genes", *Nucleic Aicds Research*, Volume 33, Issue suppl 1, 2005, Pages 139 - 140.

[84] G. Storz , S. Gottesman, "Versatile roles of small RNA regulators in bacteria", *The RNA world, 3rd ed.*, 2006, Pages 567-594.

[85] Thomas Sterling, "Beowulf Cluster Computing with Linux, edited by", *The MIT Press*, 2002.

[86] I. Jr. Tinoco, O. C. Uhlenbeck, M. D. Levine, "Estimation of secondary structure in ribonucleic acids", *Nature* , Volume 230, Issue 5293, 9 April 1971, Pages 362-367.

[87] I. Jr. Tinoco, P. N. Borer, B. Dengler, M. D. Levine, O. C. Uhlenbeck, D. M. Crothers, J. Gralla, "Improved estimation of secondary structure in ribonucleic acids", *Nature New Biology*, Volume 246, Issue 150, 14 November 1973, Pages 40-41.

[88] J. E. Tabaska, R. B. Cary, H. N. Gabow, G. D. Stormo, "An RNA folding method capable of identifying pseudoknots and base triples", *Bioinformatics* , Volume 14, Issue 8, 1998, Pages 691-699.

[89] F. Tahi, M. Gouy, and M. Regnier, "Automatic RNA secondary structure prediction with a comparative approach", *Computers and Chemistry* , Volume 26, Issue 5, July 2002,Pages 521-530.

[90] F. Tahi, S. Engelen, M. Regnier, "A Fast Algorithm for RNA Secondary Structure Prediction Including Pseudoknots", *Bioinformatic and Bioengineering, IEEE International Symposium* , 26 March 2003, Pages 11-17.

[91] B. J. Tucker , R.R. Breaker , "Riboswitches as versatile gene control elements", *Current Opinion in Structral Biology* , Volume 15, Issue 3, June 2005, Pages 342-348.

[92] R. Tyagi , D. H. Mathews, "Predicting helical coaxial stacking in RNA multibranch loops", *RNA* , Volume 13, Issue 7, July 2007, Pages 939-951.

[93] G. Tan, N Sun, G. R. Gao, "A Parallel Dynamic Programming Algorithm on a Multi-core Architecture", *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures* , June 2007, Pages 135-144.

[94] G. Tan, N Sun, G. R. Gao, "Improving Performance of Dynamic Programming via Parallelism and Locality on Multicore Architectures", *IEEE Transactions*

on Parallel and Distributed Systems , Volume 20, Issue 2, February 2009 , Pages 261-274.

[95] Y. Uemura , A. Hasegawa , S. Kobayashi , Yokomori, "Grammatically modeling and predicting RNA secondary structures", *Proceedings of the Genome Informatics Workshop*, 1995, Pages 67-76.

[96] P. Walter , G. Blobel, "Signal recognition particle contains a 7S RNA essential for protein translocation across the endoplasmic reticulum", *Nature*, Volume 299, Issue 5885, 21 October 1982, Pages 691-698.

[97] A. E. Walter , D. H. Turner , J. Kim , M. H. Lyttle , P. MÃijller, D. H. Mathews , M. Zuker, "Coaxial stacking of helixes enhances binding of oligoribonucleotides and improves predictions of RNA folding", *Proceedings of the National Academy of Sciences of the United States of America* , Volume 91, Issue 20, 27 September 1994, Pages 9218-9222.

[98] J. D. Watson, F. H. C. Crick, F. H. C, "Molecular structure of nucleic acids: A structure for deoxyribose nucleic acid." *Nature*, Volume 171, 25 April 1953, Pages 737Ð738.

[99] C. R. Woese, N. R. Pace, B. C. Thomas, "Probing RNA structure, function, and history by comparative analysis", *The RNA World* , 1993, Pages 91-117.

[100] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, K. A. Yelick, "The Potential of the Cell Processor for Scientific Computing", *Proceedings of the 3rd conference on Computing frontiers* , Volume 6, 2006, Pages 9-20.

[101] L. Wu , J. G. Belasco, "Let me count the ways: Mechanisms of gene regulation by miRNAs and siRNAs", *Molecular Cell* Volume 29, Issue 1, 18 January 2008, Pages 1-7.

[102] A. Wirawan, C. K. Kwoh, B. Schmidt, "Parallel DNA sequence alignments on the Cell Broadband Engine", *Parallel Processing and Applied Mathematics*, Volume 4967/2008, Pages 1249-1256.

[103] I. K. Yanson, A. B. Teplitsky, L. F. Sukhodub, "Experimental Studies of Molecular Interactions Between Nitrogen Bases of Nucleic Acids", *Biopolymers*, Volume 18, Issue 5, May 1979, Pages 1149-1170.

[104] M. Zuker, P. Stiegler, "Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information", *Nucleic Acids Research*, Volume 9, Issue 1, 10 January 1981, Pages 133-148.

[105] M. Zuker, D. Sankoff, "RNA secondary structures and their prediction", *Bulletin of Mathematical Biology*, Volume 46, Issue 4, 1984, Pages 591-621.

[106] M. Zuker, "Computer prediction of RNA structure.", *Methods Enzymol*, Volume 180, 1989, Pages 262-288.

[107] M. Zuker, "On finding all suboptimal foldings of an RNA molecule", *Science*, Volume 244, Issue 4900, 7 April 1989, Pages 48-52.

[108] M. Zuker, D.H. Mathews, D.H. Turner, "Algorithms and thermodynamics for RNA secondary structure prediction - A practical Guide", *RNA Biochemistry and Biotechnology* , 1999, Pages 11-43.

[109] W. Zhou, D. K. Lowenthal, "A Parallel, Out-of-Core Algorithm for RNA Secondary Structure Prediction", *Parallel Processing ICPP* , Volume, 16 October 2006, Pages 74-81.

[110] "FHD(Eke) van Batenburg: homepage of PseudoBase", `http://www.ekevanbatenburg.nl/PKBASE/PKB.HTML`.

[111] "Crowd-sourced Information on RNA Secondary Structure", `http://en.wikipedia.org/wiki/Secondary_structure`.

[112] "CELL Broadband Engine Resource Center", `http://www.ibm.com/developerworks/power/cell/`.

[113] "tRNAdb transfer RNA database", `http://trnadb.bioinf.uni-leipzig.de/`.

[114] "RCSB The Protein Data Bank", `http://www.rcsb.org/pdb`.

[115] "NDB The Nucleic Acid Database", `http://ndbserver.rutgers.edu/`.

[116] "The Comparative RNA Web (CRW) Site:", `http://www.rna.ccbb.utexas.edu/`.

Author's Publications

[1] S. P. T. Krishnan, Sim Sze Liang, and Bharadwaj Veeravalli, "Towards High-Performance Computing for Molecular Structure Prediction using IBM Cell Broadband Engine - an implementation perspective", in *Eighth Asia-Pacific Bioinformatics Conference (APBC 10)*, Bangalore, India, January 18-21, 2010.

[2] S.P.T. Krishnan, Mushfique Junayed Khurshid, and Bharadwaj Veeravalli, "A Matrix Algorithm for RNA Secondary Structure Prediction", in 5th *IAPR International Conference, PRIB 2010*, Nijmegen, The Netherlands, September 22-24, 2010

[3] S. P. T. Krishnan, Sim Sze Liang, and Bharadwaj Veeravalli, "Towards High-Performance Computing for Molecular Structure Prediction using IBM Cell Broadband Engine - an implementation perspective", in *BMC Bioinformatics, Volume 11, (Suppl 1): S36, January 2010.*

[4] S. P. T. Krishnan, Bharadwaj Veeravalli, "Performance Characterization and Evaluation of Biological Structure Prediction Algorithms on Homogenous

and Heterogeneous Multi-core Architectures", under review by *Journal of Parallel and Distributed Computing*.

[5] S. P. T. Krishnan, Bharadwaj Veeravalli, "Case Study of Google App Engine Suitability for Developing HPC Applications - Challenges and Opportunities", under review by *IEEE Transaction on Parallel and Distributed Systems*.