MULTI-STAGE POLYPEPTIDES COMPARISON IMPROVEMENTS FOR A PARALLEL PROTEIN INTERACTION PREDICTION ENGINE

by

Christopher North

A thesis submitted to the Faculty of Graduate Studies

Carleton University

in partial fulfillment of the requirements

for the degree of Master of Computer Science in the

Ottawa-Carleton Institute for Computer Science

Your file Votre référence
ISBN: 978-0-494-27010-3
Our file Notre référence
ISBN: 978-0-494-27010-3

Abstract

The discovery of protein-protein interactions is fundamental to the understanding of the biological processes that occur within organisms. Exhaustively testing all possible protein-protein interactions in an organism in a laboratory is both a time consuming and expensive endeavor. Computational approaches have been developed to alleviate these costs, however, these approaches typically have much higher rates of false positives.

Protein-Protein Interaction Prediction Engine (PIPE), developed by Pitre et al., is a novel computational approach that offers acceptable rates of false positives, however, it is very computationally intensive. Running PIPE on the entire yeast proteome is estimated to take on the order of hundreds of years. This thesis presents a number of improvements to the PIPE algorithm which result in a four-orders of magnitude performance improvement. These enhancements make it possible to run the PIPE algorithm on the entire yeast proteome in less than one week.

Acknowledgments

I would like to thank my supervisor, Dr. Frank Dehne, for his advice and support. As well, I would like to thank other members of the Carleton University School of Computer Science Bioinformatics Research Group for their insight, suggestions, and advice throughout my research. In particular, I would like to thank Sylvain Pitre, Dr. Ashkan Golshani, Dr. James Green, and Dr. Michel Dumontier.

I would like to thank Luc Ostiguy at Cray Inc. for years of advice and mentoring, and for instilling in me the belief that high performance computing is about more than just cobbling together a bunch of PCs.

Finally, I would like to thank my spouse, Megan Dewar, for her years of support, motivation, and inspiration, and for encouraging me to pursue graduate studies and research.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation for studying protein interactions

The study of protein molecules and their associations and interactions with other molecules is fundamental to the understanding of the biological processes that occur within organisms. The cellular functions that are present in all organisms are the result of a complex network of protein-protein interactions. For instance, transcription (and subsequently, translation) occurs when RNA polymerase binds to a specific region on DNA. In order to decipher the internal workings of biological structures, it is necessary to understand how proteins interact with one another.

The study of protein-protein interactions has a direct impact on the design and development of new pharmaceuticals [2]. Most pharmaceuticals work by targeting

specific cellular functions and proteins. By improving our knowledge of protein-protein interactions, pharmacologists are able to produce better drugs with fewer side-effects.

Since the early 1990s, major international and multi-disciplinary efforts have taken place with the goal of sequencing the entire human genome as well as the genomes of other organisms. This work has lead to significant advances in the development of specialized equipment and methods supporting high-throughput gene sequencing. Between 1994 and 2004, over 180 organisms were fully sequenced. The most notable of these efforts, the Human Genome Project, was coordinated by the U.S. Department of Energy and the National Institutes of Health and took over 13 years to complete [30]. The resulting data was surprising for two reasons. First, the number of genes in the human genome is much less than originally believed (30000 as opposed to 100000). Second, humans share many genes with other organisms. The laboratory mouse (*Mus musculus*) and human (*Homo sapiens*) both have approximately 30000 genes, of which 85% are identical [31].

By studying known protein-protein interactions and discovering new interactions, scientists develop a better picture of the dependencies and interactions that genes have with one another. This knowledge also assists in reconstructing the metabolic and signaling pathways of cells and provides insight into the co-evolution of genes [6]. These are all crucial steps in determining how proteins work together to bring about

life.

1.2 Determining protein-protein interactions

On April 14, 2003 the Human Genome Project was declared a success with the mapping of 99% of the human genome with an accuracy of 99.99%. This marked the end of the genomic era and left scientists trying to figure out how to make sense of all the data produced. For every organism sequenced, scientists now have a list of nearly all the proteins in the organism and, consequently, a list of all possible protein-protein interactions. However, given that even small organisms such as yeast (*Saccharomyces cerevisiae*) have over 6300 proteins, exhaustively testing all possible protein pairs for interaction is not practical using present-day experimental or computational methods (there are on the order of $6300^2/2 = 19845000$ possible protein pairs for yeast). Sections 2.1 and 2.2 provide more information on common methods used for determining protein-protein interactions.

1.3 A New Computational Approach

In 2005, Pitre et al. proposed a new computational approach for finding protein-protein interactions. A working model of this approach, called Protein-Protein Interaction Prediction Engine (PIPE), has been used to discover several novel interactions

for the yeast organism, leading to the identification of a previously unknown complex [29].

Unfortunately, PIPE suffers from a common problem of many other computational and experimental methods; it takes a long time to perform each prediction. PIPE takes on average 115 minutes of processing time for each protein pair. Even with a supercomputer with thousands of processors, the runtime for testing all possible protein pairs for yeast is on the order of hundreds of years. Unless the PIPE runtime is reduced to allow the exhaustive testing of all possible protein pairs in an organism, it offers few benefits over other computational and experimental approaches.

1.4 Contributions

This thesis presents improvements made to the PIPE algorithm. These improvements focus on reducing the computational cost of individual window comparisons. The three main improvements made to the PIPE implementation are: using a digital alphabet for the protein sequence representations, applying a "rolling comparison" approach to the fixed-length window comparisons, and performing the pre-computation of all possible window comparisons before the protein-protein interaction prediction stage. These improvements have resulted in a four-orders of magnitude speedup making it possible to test all possible protein pairs in yeast. The resulting data has lead to a more complete understanding of the PIPE algorithm, including the discovery of

areas for improvements and shortcomings not previously known.

1.5 Section Synopsis

The rest of the thesis is organized as follows. Chapter 2 provides background on the relevant biological and computational theories and methods used in determining protein-protein interactions. Chapter 3 presents the original PIPE algorithm. Chapter 4 presents the improvements made to the PIPE algorithm and discusses the computational impact of these improvements. Chapter 5 presents the results obtained using the improved version of PIPE. Chapter 6 discusses future work and new directions for the PIPE algorithm including the requirements necessary for predicting protein-protein interactions in other organisms. Finally, Chapter 7 provides a summary of the work presented in this thesis.

Chapter 2

Background

The terms bioinformatics and computational biology refer to the interdisciplinary fields that employ aspects of mathematics, physics, computer science, engineering, chemistry and biology to advance our understanding of biological structures. In an attempt to distinguish the two, the U.S. National Institutes of Health has come up with the following definitions.

> *Bioinformatics:* Research, development, or application of computational tools and approaches for expanding the use of biological, medical, behavioral or health data, including those to acquire, store, organize, archive, analyze, or visualize such data. [21]
>
> *Computational Biology:* The development and application of data-analytical

> and theoretical methods, mathematical modeling and computational simulation techniques to the study of biological, behavioral, and social systems. [21]

This chapter provides a background on the fields of experimental and computational biology as they pertain to the discovery of protein-protein interactions. In addition, this chapter provides an introduction to select areas of computer science and high performance computing.

2.1 Experimental Methods

Laboratory experiments typically seek out three classes of protein-protein interactions. The first class are direct physical interactions and occur when two proteins physically interact with one another. The second class are indirect physical interactions and occur when two proteins take part in the same complex, but do not physically interact. The third class are functional interactions and occur when two proteins are shown to take part in the same cellular function but do not physically interact or belong to the same complex.

Depending on the class of interactions sought, there are different experimental methods employed [12, 15, 19, 20, 38, 43]. Unfortunately, none of the experimental methods available today are well suited for the exhaustive testing of all possible protein-protein interactions in a given organism. Not only are these methods very

time consuming and resource intensive, they also suffer from relatively high rates of false positives. In small scale experiments it is not uncommon for the same test to be performed multiple times in order to verify the results. This approach to achieving a higher level of confidence is not feasible in large-scale exhaustive experiments. The effects of even a moderate level (say, 1%) of false positives is compounded by the small ratio of real interactions to total possible interactions. In yeast it is estimated that there are between 16000 – 26000 actual protein-protein interactions, however, there are just under 20 million total possible protein-protein interactions [17]. With a 1% false positive rate, the experiment would produce over 200000 incorrect interactions far outweighing the 16000 – 26000 correct interactions.

2.2 Computational Methods

Computational methods are frequently used in the analysis and comparisons of protein sequences and structures. Computational methods can typically be assigned to one of three categories. The first category examines amino-acid sequences in order to characterize and find similarities in the 3D structure, function, and expression of proteins [3, 25, 27]. The second category uses graph theoretical approaches on existing protein-protein interaction data in order to draw inferences and make predictions on the structure of the underlying interaction network [26, 40, 42]. The third category uses statistical approaches to bring together information from a variety of sources in

order to make or confirm predictions with a high level of confidence [22, 28].

Computational methods have proven to be very useful in the analysis and interpretation of experimental results. High-throughput laboratory experiments often turn to computational methods in order help reduce the number of false positives produced [23]. However, computational methods alone have not garnered much interest in the large scale search of all possible protein-protein interactions. There are two reasons for this. The first reason is that computational methods often implement very complex mathematical algorithms requiring significant computation time and resources. Performing this computation on all possible protein-protein interactions may well be prohibitive even when using a modern high performance computing architecture. The second reason is that computational methods suffer from the same (if not worse) high rates of false positives as experimental methods.

2.3 High Performance Computing

High Performance Computing (HPC) is a branch of computer science that focuses on achieving maximum performance (measured in a variety of ways) in a given application. The theories and methods employed in this pursuit typically involve processor technologies, memory technologies, parallel architectures, and parallel programming paradigms. Achieving performance improvements in an application necessitates both single and multi-processor optimizations and may also necessitate the development

and implementation of new algorithms that are computationally more efficient.

To achieve performance optimizations in an application it is necessary to have a very good understanding of the underlying algorithm and its implementation. This includes the overall flow of the application, processing requirements and characteristics, memory access patterns, inter-processor communications, and I/O. By identifying these elements, a programmer should be able to locate the bottlenecks in the application and identify potential areas for improvement. If this level of analysis and improvements does not yield satisfactory results, it is likely that a change in algorithm is required. This may involve the parallelization of the application or a complete redesign of the algorithm to take advantage of the underlying hardware and avoid application bottlenecks.

When designing and optimizing an algorithm, one must not only be cognizant of the theoretical performance costs and bottlenecks, but also take into consideration the target computer architecture. Depending on the architectures available, certain optimization approaches may result in better improvements. An example of this is the gains achieved on vector processors from pipelining. In contrast, cache-based microprocessors do not benefit nearly as much from this type of optimization (especially for gather/scatter type operations).

2.3.1 Processor Technologies

Over the past 20 years, the performance gap between commodity microprocessors and specialized processing technologies has decreased substantially. For instance, in 1975 the Cray-1 vector processor operated at 80MHz and could perform one floating-point calculation every clock cycle resulting in 80 million floating point operations per second. In the same year, Intel's 8008 processor operated at 108 kHz and could perform a mere one hundred floating point operations per second [11]. In contrast, today's microprocessors can outperform vector processors in both clock frequency and floating point operations per second and only lag behind vector processors in the area of sustained memory bandwidth.

Given the exponential growth in commodity CPUs speed, coupled with their low-cost and general availability, microprocessors have now become the processor of choice for many high performance computing vendors. In the November 2006 edition of the TOP500 supercomputers list, 98.6% of the systems were comprised of cache-based microprocessors. The remaining 1.4% were vector-based systems. This does not mean that specialized processing architectures have no future in the high performance computing world. Vector processors continue to perform very favorably on applications that contain large pipelined loops. There has also been interest in the use of field programmable gate arrays (FPGAs) and graphics processing units (GPUs) for use as specialized co-processing within a traditional HPC environment [16, 24]. More

recently, there has been much discussion and excitement regarding the STI Cell processor developed by Sony, Toshiba, and IBM. This processor was initially developed for use in the PlayStation3 computer gaming system and has been shown to outperform even the fastest microprocessors on single-precision floating point calculations [39]. In the past three years there has been growing evidence that future HPC architectures will be more heterogeneous, comprising of microprocessors together with commodity special purpose co-processors such as FPGAs, GPUs, and Cell processors [14].

As mentioned earlier in this section, when designing and optimizing an algorithm one must take into consideration the target architecture. Currently, the main HPC architecture available at Carleton University School of Computer Science is the HPCVL Linux Cluster. This cluster is made up of 64 servers, each comprised of 2 Intel Pentium 4 Xeon processors. Table 1 provides a summary of the key architectural features of the Intel Xeon microprocessor. Interprocessor communications of this cluster are handled by a Cisco Catalyst 6509 switch. This system has a maximum bandwidth of 116.425 MB/sec with a minimum ping-pong latency of 42.238 microseconds as measured using the High Performance Computing Challenge benchmark suite (see http://icl.cs.utk.edu/hpcc/ for more information).

	Intel Xeon (32-bit)
Clock Speed	$\geq$ 1.7 GHz
FPU Units	1 FMUL/FADD 1 FSTORE
Integer Units	2 Double Pumped, 1 Slow
Load / Store	1 Load / 1 Store
SIMD	1 x SSE/SSE2

Table 1: Architectural features of the Intel Xeon processor

2.3.2 Memory Technologies

Almost all memory technologies used today are based on either dynamic random access memory (DRAM) or static random access memory (SRAM). DRAM are charge-based devices in which each bit is stored in a capacitor. Since the charge in the capacitor can leak away over time, DRAM must be continually refreshed in order to maintain a given state. SRAM are gate-based devices with each gate made up of several transistors. SRAM maintains its state as long as there is power flowing through the device. SRAM tends to be much faster than DRAM, and consequently, is more expensive and produces more heat [11].

Reducing memory latency has become one of the most important aspects in achieving performance on a modern microprocessor. In the early 1980s, the main memory access time was shorter than the processor's clock frequency. This meant that a microprocessor could connect directly to the main memory without incurring processor delays caused by memory access. However, throughout the last two and a half decades,

microprocessor clock frequencies have increased much more rapidly than memory access speeds. It is generally accepted that if the imbalance between memory and processor speeds continues to grow, application run-time on future microprocessors will be dominated by memory access time [4, 41].

In order to keep up with increasing processor speeds, microprocessor architectures have had to incorporate various methods to reduce the impact of memory access delays. Unlike vector processors, which exploit pipelining in order to minimize the cost of each memory access, modern microprocessors employ a memory hierarchy with the goal of keeping the data being operated on as close to the processor as possible. This hierarchy includes the registers, L1/L2/L3 cache, main memory, and virtual memory. Table 2 provides a summary of the memory hierarchy for the Intel Xeon microprocessor (dual-processor model).

2.3.3 Parallel Computing

Parallel computing refers to the execution of a given application on multiple processors. The main goal of parallel computing is to reduce the total runtime of an application by splitting the problem into smaller sub-problems that can each be solved concurrently on different processors. Another goal of parallel computing is to scale up the application problem size beyond what is possible to solve on a single processor. Flynn's taxonomy, which has been in use since 1966, is used to classify parallel

	Intel Xeon (32-bit)
General Purpose Registers	8
L1 Data Cache	8 KB
L1 Instruction Cache	12,000 μ−ops (~20 KB)
L1 Cache Latency	2 cycles
L1 TLB	64 entries
L2 Cache	512 KB
L2 Cache Width	256 bit
L2 Cache Latency	9 - 20 cycles
Max Memory Bandwidth	4.2 GB/s
Memory Latency (Measured)	102 ns

Table 2: Memory hierarchy of the Intel Xeon processor

computers and consists of four distinct categories.

1. **Single Instruction, Single Data (SISD)**: This category refers to serial (non-parallel) computers. Only one instruction is executed at a time, and only one piece of data is being operated on at a time. This category is used to describe most single CPU computer systems.

2. **Single Instruction, Multiple Data (SIMD)**: In a SIMD system all processors execute the same instruction at the same time, however, each processor can operate on different data elements. These systems require synchronization between processors. This category includes systems comprised of processor arrays (e.g. Connection Machine CM-2, and Maspar MP-1) and vector pipelines (e.g. Cray C90, NEC SX-2).

3. **Multiple Instruction, Single Data (MISD)**: A MISD system has the same data stream fed into multiple processors. Each processor performs an independent set of instructions on the data simultaneously. This is an experimental parallel architecture with few actual implementations. The Carnegie-Mellon C.MMP computer is one example.

4. **Multiple Instruction, Multiple Data (MIMD)**: In a MIMD system the processors are free to execute independent instruction streams on independent data elements. Most modern parallel computers fall into this category. This includes massively parallel processing systems (MPPs), symmetric multiprocessor systems (SMPs), and cluster-based systems.

One of the biggest challenges faced in parallel computing is overcoming the overhead associated with the parallelization of the application. This overhead is caused by several factors and can significantly impact the speedup and scaleup that can be achieved on a parallel computer.

A first cause of overhead is the application startup time. Parallel computers containing multiple operating systems (such as clusters and MPPs) typically have longer startup times than systems with single operating systems since the application needs to launch multiple times (once per operating system). Depending on the number of processes that need to be launched and the runtime of the application, the job startup may make up a considerable portion of the total runtime.

A second cause of overhead is due to the added complexity required to parallelize the application. An application running in parallel may be significantly more complex than an application running on a SISD system. This added complexity can typically be attributed to changes in the underlying algorithm and/or the need to split the problem into smaller sub-problems. Depending on the changes required to run the algorithm on a parallel system, the efficiency of the application may be diminished (although overall runtime may still be reduced).

A third cause of overhead can be attributed to the processor synchronization and inter-processor communication that may occur throughout the parallel application. When processors need to interact with one another (either through a synchronization point, remote memory access, or message passing), there is a processing delay incurred as a result of this communication. On most parallel systems, the underlying communication network between processors is much slower than the local memory management system. If the parallel application does a significant amount of inter-processor communication, the network latency and bandwidth may adversely effect the potential speedup of the parallel application. This is especially true for clusters comprised of commodity networking hardware [36].

A forth factor that may limit a parallel algorithm's speedup is expressed by Amdahl's law. This principle states that the potential speedup that can be achieved by a parallel program is a function of the portion of the program that can be parallelized.

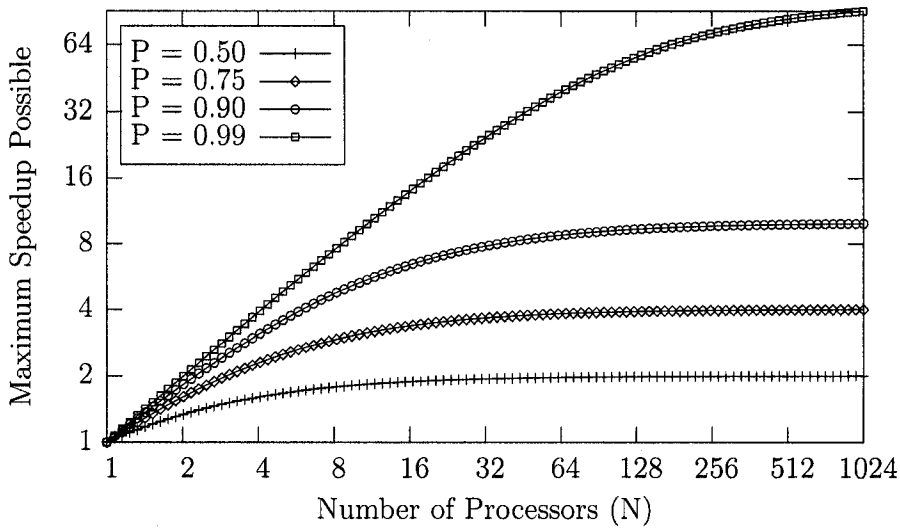

Figure 1: Amdahl's Law — Maximum Speedup Possible for various P values

If P is the percentage of the program that can be parallelized, S is the percentage of the program that cannot be parallelized ($S = 1 - P$) and N is the number of processors, then the maximum speedup possible is $(P/N + S)^{-1}$. Figure 1 shows the effects of Amdahl's law for various values of P. Log scales for both x and y axes are used to better illustrate the behaviour of the function for small P values.

2.3.4 HPC Software Tools

Given the myriad hardware technologies and architectures used in the high performance computing environment, it is necessary to have a powerful set of software tools that will assist the programmer in adapting their application to the given parallel architecture. In order to achieve maximum performance on a HPC system, the

programmer must effectively employ the available parallel programming languages and performance analysis tools. In doing so, the programmer will become aware of any hardware or software bottlenecks that exist and can take appropriate action to correct these problems.

Parallel Programming Paradigms

In the past ten years, the high performance computing environment has undergone a dramatic change in system architectures and programming environments. Due to the tremendous improvements in commodity microprocessors, today's high performance computing environment is lead primarily by clustered architectures comprising of multiple distinct nodes each with their own processor, memory and, in most cases, operating systems [36]. This phenomenon has had a direct impact on the parallel programming paradigms employed. As clustered architectures have become more prevalent, the shared memory programming model has given way to the distributed memory model. This transition has improved the performance of clustered systems by placing the responsibility for message passing in the hands of the programmer. Consequently, applications written using the distributed programming model are more complicated and difficult to maintain than their shared memory counterparts. Below we discuss these programming models in detail.

The *shared memory programming model* provides a global shared address space

that is visible to all processors. Reading and writing to shared memory locations is performed automatically within expressions and assignments. The simplicity of shared memory access results in code that is very similar to the single processor version of the code. Synchronization and locking mechanisms are provided to help the programmer avoid race conditions. One weakness of the shared memory programming model is that it does not allow exploitation of data locality. A given processor cannot distinguish between its local memory and memory that is on a remote processor. OpenMP is an example of the shared memory programming model [32]. Shared memory programming languages are typically only available on Symmetric Multi-Processing systems (SMPs) and Non-Uniform Memory Access (NUMA) architectures.

The *distributed memory programming model* uses message passing to share data between processes. The programmer explicitly controls data sharing and work distribution using, in most cases, two-sided communication between processors. This allows the programmer to control the amount of message passing required based on the speed of the underlying network which can improve the overall efficiency of the algorithm. To achieve this efficiency, the programmer must re-design the algorithm using the message passing paradigm. As a result, distributed memory model programs tend to be more complicated than their serial or shared memory model counterparts. Implementations of the distributed memory programming model are typically optimized

for large buffered data transfers, and consequently there is significant communication overhead for small transactions. MPI is an example of the distributed memory programming model [18, 37]. The distributed memory programming model is geared towards clustered systems and Massively Parallel Processing systems (MPPs).

The *distributed shared memory programming model* is similar to the shared memory programming model in that it provides a global shared address space visible by all processors. It also provides for automatic remote memory access within expressions and assignments. It differs from the shared memory model in that the global address space is partitioned in such a way that every shared memory location has affinity to a particular processor. This enables the exploitation of data locality and also allows the programmer to control the amount of inter-processor communications depending on the speed of the underlying network. Unified Parallel C (UPC) and Co-Array Fortran are both examples of the distributed shared memory programming model [5, 13]. Despite their benefits, distributed shared memory programming languages have not yet been widely adopted within the HPC environment. There are two likely reasons for this. The first reason is the lack of optimized compilers and tools for these languages. Proponents of this model suggest that this problem will be addressed over time as these languages become more mature and gain popularity. The second reason is due to the saturation of MPI within the HPC environment. After the MPI standard was released in 1992, it quickly replaced PVM as the de facto choice for

parallel programming languages. The near universal adoption of MPI as a parallel programming language makes it very difficult for new languages to be embraced by the HPC user community.

Performance Analysis Tools

Once a programmer has chosen the HPC architecture to use and has implemented their algorithm, it is time to begin analyzing the performance of the program to determine areas for improvement. The key to identifying these areas for improvement is to understand what portions of the program consume the majority of the runtime and why. If the programmer is able to identify specific software and/or hardware deficiencies, they can focus their effort on these particular aspects of the program. There are generally three techniques used to identify these performance deficiencies: time-domain sampling, application instrumentation, and hardware performance counters [8].

Time-domain sampling is an approach where, as the program is being run, it is queried at regular time intervals (e.g. every 10 milliseconds) and the address of the instruction currently being executed is recorded. This address can be used to determine which function was being executed at a given point in time. After the program has completed execution, this information, combined with the total application runtime, is used to determine how much time was spent in each function.

It is assumed that, given a regular interval period, the amount of time spent in a given function is proportional to the number of samples collected within that function. The advantage of time-domain sampling is that it is a simple approach that significantly impact the program execution time. The problem with this approach, however, is that most time-domain sampling tools use imprecise virtual timers that are updated only after the operating system scheduler is run. If the application being profiled contains functions that trigger the scheduler, results from the profile will be biased towards these functions. In general, this approach provides the programmer with a rough estimate of the percentage of time is spent in each function.

Application instrumentation is similar to time-domain sampling in that it uses timers to determine the percentage of the runtime spent in each function. However, unlike time-domain sampling, application instrumentation measures and records this information explicitly by inserting function calls throughout the application. Automatic application instrumentation is usually performed at function entry and exit points to determine how much time is spent in each function. Block level instrumentation is also possible by manually instrumenting the code before and after each target block. Unlike time-domain sampling, application instrumentation does not suffer from scheduler-induced function bias. Depending on the architecture, application instrumentation tools may even use high resolution hardware counters to more

accurately measure function/block entry and exit points. Another benefit of application instrumentation over time-domain sampling is that the profile produced includes time spent not executing the program (e.g. waiting for an I/O request to complete). This information may be very useful in determining application bottlenecks. The main disadvantage of application instrumentation is that it adds additional overhead to the execution of the application. Every entry and exit point will call an instrumentation routine. Depending on the runtime of each function and the number of times the function is called, this instrumentation overhead can produce results that are skewed towards functions that have short runtime but are called frequently. In general, application instrumentation is more powerful than time-domain sampling since it more accurately measures the amount of time spent in each function and allows the programmer to manually instrument specific areas of the code that may require a more detailed analysis.

Hardware performance counters are a set of registers available on all modern microprocessors that are capable of counting various hardware events that occur throughout the execution of a program. The specific events available differ from one architecture to another but generally cover the following categories: branching information, arithmetic instructions (both floating point and integer), hardware interrupts, L1 and L2 data/instruction cache access, load/store instructions, translation lookaside buffer (TLB) access, instructions issued/completed, and total cycles. Performance

counter information is generally collected using an interface library. Libraries such as Performance API (PAPI) record performance counter information by instrumenting the program before and after areas of interest. Performance counters are very powerful because they assist the programmer in determining the hardware behaviour corresponding to specific portions of the program.

Chapter 3

PIPE

In 2005, Pitre et al. [29] proposed a new computational approach for finding protein-protein interactions. This approach is based on the hypothesis that there are a finite number of polypeptide sequences that are responsible for mediating the interaction between proteins. This approach relies on the re-occurance of fixed-length polypeptide sequences as well as known protein-protein interaction data to determine the likelihood of an interaction between two given proteins. The computer program that implements this algorithm is called Protein-Protein Interaction Prediction Engine (PIPE). By using a known protein-protein interaction network which represents relationships in which we have high confidence, and carefully tuning the program parameters, PIPE has demonstrated rates of success similar to most commonly used experimental methods [29].

The PIPE algorithm works as follows. Suppose you are evaluating the likelihood of an interaction between two target proteins, A and B. Suppose also that you know from experimental results that two other proteins, C and D, are known to interact. If protein A has subsequences similar to those of protein C and protein B has subsequences similar to those of protein D, then it is possible that proteins A and B also interact if these common subsequences are responsible for the interaction between proteins C and D. To determine the likelihood that A and B interact, we consider subsequences of A and B. For each pair of subsequences, we count the number of protein pairs that are known to interact that contain similar subsequences.

3.1 Algorithm Details

The PIPE algorithm predicts the likelihood of an interaction between two proteins by measuring the number of co-occurrences of fixed-length polypeptides with protein pairs that are known to interact. For the PIPE algorithm, and the subsequent optimized PIPE algorithm, a *window* is a fixed-length polypeptide made up of a consecutive string of amino-acids. In other words, a window is a fixed-length subsequence of a protein sequence. In the PIPE programs, the window length is defined by the global constant, L. In the preparatory work for the original PIPE implementation, much work went into determining an optimal value for L. We use this value of L in the optimized PIPE algorithm. Windows are specified by a unique starting point

within protein sequences and may overlap in at most $L-1$ amino acids. Therefore, given a protein sequence containing N amino acids, there are a total of $N-L+1$ distinct windows. Figure 2 demonstrates the relationship between a protein sequence S containing N amino acids $(a_1, a_2, \ldots, a_N)$ and the corresponding windows of length $L = 4$ $(W_1, W_2, \ldots, W_{(N-L+1)})$.

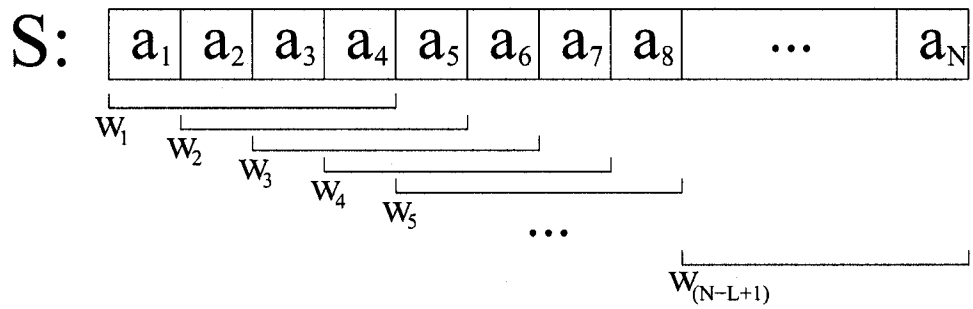

Figure 2: Relationship between protein sequence, amino acids, and windows

Two windows are said to be *similar* if there is a positive likelihood that they have evolved from the same ancestor. The PIPE algorithm tests whether two windows are similar by using the PAM120 percent accepted mutation matrix [1]. This matrix is a commonly accepted standard for determining the likelihood that one amino acid may have mutated into another. When comparing two windows, the PAM120 matrix is applied to corresponding amino acid pairs within each window and the resulting values are summed together to produce an overall score for the two windows. If this score is above the pre-computed threshold, the two windows are said to be similar. In the development of the original PIPE algorithm, Pitre et al. performed an analysis of these scores and determined an appropriate threshold value resulting in an acceptable

level of confidence [29]. For windows of length 20, a threshold value of 35 was used.

The PIPE algorithm uses known protein-protein interaction information to predict new interactions. The protein-protein interaction information is presented to the PIPE algorithm in the form of an interaction list. This list is easy to interpret as an interaction graph where the nodes represent the proteins and the edges represent interactions. It is assumed that if protein A has been shown to interact with protein B, then protein B also interacts with protein A, thus the edges of the graph are undirected. Table 3 and Figure 3 demonstrate how the known interaction lists are represented as a graph. The term *neighbour* will be used to describe two nodes in a graph that are directly connected to one another.

Protein	Interaction List
A	$\{C, D, E\}$
B	$\{C\}$
C	$\{A, B, D\}$
D	$\{A, C\}$
E	$\{A\}$
G	N/A
F	N/A

Table 3: Table of interaction information

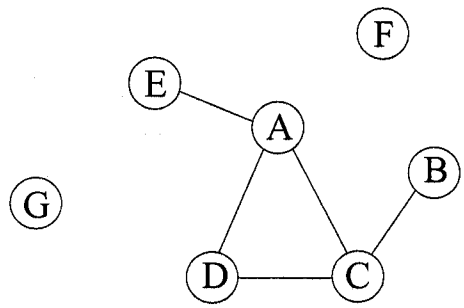

Figure 3: Interaction graph

With the required terminology defined, the PIPE algorithm can be described in pseudo-code; shown in Figure 4. This algorithm counts the number of times that pairs of windows from two target proteins are similar to those in known interacting proteins. These counts are represented in a matrix H. $H[i][j]$ is the number of times windows i and j from the first and second target proteins are similar to windows in proteins that

are known to interact. The two-dimensional matrix H can be represented graphically as a three-dimensional histogram where the X and Y axes represent the windows of the first and second target proteins and the Z axis is the number of times similar windows were found in known interacting pairs. Figure 5 is an example of the three-dimensional histogram produced for target proteins YGR261C and YBR288C.

Definitions:
Let $A :=$ the first target protein
Let $B :=$ the second target protein
Let $X_k := k^{th}$ window of a given protein X
Let $\mathcal{S} :=$ the list of all proteins
Let $P \longleftrightarrow Q$ denote that proteins P and Q are neighbours (interact)
Let $X_\ell \sim Y_m$ denote that windows X_ℓ and Y_m are similar

Algorithm:

```
For every window A_i ∈ A {
    For every protein X ∈ S {
        If ∃ k such that X_k ~ A_i {
            For every protein Y ∈ S such that X ⟷ Y {
                For every window B_j ∈ B {
                    If ∃ m such that Y_m ~ B_j {
                        H[i][j] := H[i][j] + 1
                    }
                }
            }
        }
    }
}
```

Figure 4: The original PIPE algorithm

In Figure 5, the peak that can be seen near $x = 780$, $y = 120$ is strong evidence that these two proteins interact. In practice, the PIPE implementation reports

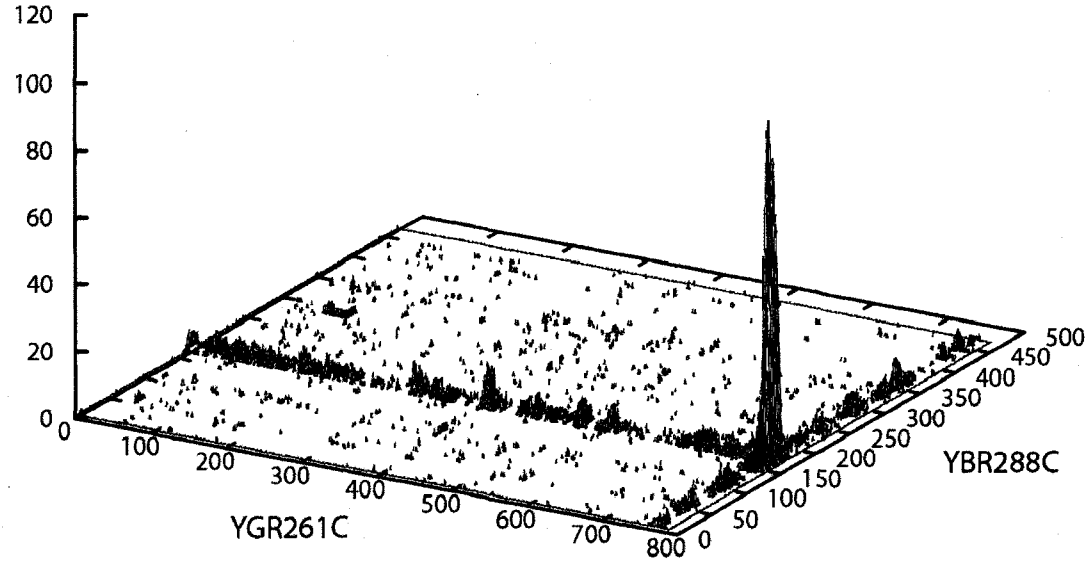

Figure 5: Three-dimensional histogram representing matrix H for YGR261C and YBR288C

an interaction if the maximum peak is greater than 10. This value was arrived at experimentally.

3.2 Parallel Algorithm

Despite attempts to produce efficient code written in C and C++, the original PIPE program has a long runtime. The runtime per target pair varies substantially and depends on the number of windows in each of the target proteins. The average runtime for the PIPE program is 115 minutes per protein pair. In order to improve this runtime, the algorithm was parallelized so that it could run on an HPC architecture. Given the simplistic nature of the PIPE algorithm, there were two obvious ways to

parallelize the program.

The first method is to distribute the running of multiple protein pairs over the number of available processors. This method does not require any changes to the code, however, it is only beneficial if there are at least as many protein pairs as there are processors. Also, given the runtime variance between protein pairs, some processors would complete their jobs more quickly than others resulting in poor overall utilization of the parallel system.

The second method is to parallelize over one of the $for(\ )$ loops (see Figure 4 on page 30) and have each processor perform independent iterations of the loop. The first loop, which iterates over each window of the first target protein, is the ideal loop to parallelize. Each iteration of this loop can be performed independently resulting in minimal inter-processor communications. For comparison, Figure 6 shows the original single processor algorithm while Figure 7 shows the changes to the code necessary for the second parallelization. Note that in the parallel version (Figure 7) the total number of processors is stored in the variable $nPEs$ and each processor has a unique ID stored in the variable $myPE$ (ranging from $0, 1, \ldots, nPEs - 1$).

This second parallelization method was chosen for PIPE. In order to obtain maximum efficiency from this parallel algorithm, the number of windows in protein A must be greater than the number of processors available. In an ideal world each iteration of the parallel loop would take an equal amount of time and the number of iterations

```
Let W_A := # of windows in protein A
/* For every window A_i ∈ A */
for(i = 0; i < W_A; i++) {
 /* Remaining algorithm unchanged */
}
outputResults();
```

Figure 6: Single Processor Algorithm

```
Let W_A := # of windows in protein A
/* For every window A_i ∈ A */
for(i = myPE; i < W_A; i+=nPEs) {
 /* Remaining algorithm unchanged */
}
if (myPE == 0) {
 /* Processor 0 gathers up and outputs
    results from all other processors */
}
```

Figure 7: Parallel Algorithm

would be a multiple of the number of processors. Given that neither of these are likely to be the case, the best one can hope for is that the number of iterations per processor is high. This will have the effect of smoothing out the runtime variance for each iteration; resulting in approximately the same runtime per processor.

The target HPC system used for PIPE was the HPCVL's Linux Cluster located in Carleton University School of Computer Science. This system is comprised of 64 dual-processor nodes. In order to share these resources with other users, it was decided that the PIPE program would run on only 32 processors. The organism chosen for testing the original and optimized PIPE implementation was yeast. For a window length $L = 20$, the average number of windows per protein in yeast is 455. This results in 14 iterations per processor which was considered an acceptable level of parallel efficiency.

3.3 PIPE Algorithm Cost

Despite the parallelization of the PIPE program, the average runtime for a given protein pair is approximately 4 minutes (on 32 processors). Given that there are just under 20 million possible protein pairs in yeast, it is infeasible to use the PIPE algorithm to exhaustively test all possible pairs. In order to understand the PIPE algorithm further, an analysis of the algorithmic cost is required.

3.3.1 Asymptotic Cost

Big-$\mathcal{O}$ notation is used to describe the asymptotic behaviour of an algorithm. It is an upper bound on the algorithm complexity given the algorithm inputs. For large enough inputs, the low order terms and constants associated with the exact runtime are dominated by the size of the inputs. The big-$\mathcal{O}$ cost of an algorithm is therefore simplified in terms of input sizes and can easily be used to compare the efficiency of different algorithms that perform the same task. When analyzing the asymptotic cost of the PIPE algorithm, there are three input variables to consider. Let N be the number of windows in a given protein, let M be the total number of proteins, and let P be the number of neighbours in the interaction graph for the given protein. Figure 8 presents the big-$\mathcal{O}$ analysis of each stage of the PIPE algorithm. Horizontal lines are added to distinguish blocks and their associated costs.

For every window $A_i \in A$ {	$\mathcal{O}(N)$
For every protein $X \in \mathcal{S}$ {	$\mathcal{O}(M)$
If $\exists\ k$ such that $X_k \sim A_i$ {	
For every protein $Y \in \mathcal{S}$ such that $X \longleftrightarrow Y$ {	$\mathcal{O}(P)$
For every window $B_j \in B$ {	$\mathcal{O}(N)$
If $\exists\ m$ such that $Y_m \sim B_j$ {	$\mathcal{O}(1)$
$H[i][j] := H[i][j] + 1$	
} } } } } }	

Figure 8: Big-$\mathcal{O}$ analysis of the original PIPE algorithm

This analysis yields an asymptotic cost of $\mathcal{O}(N^2 \times M \times P)$ for the PIPE algorithm. This is truly an asymptotic bound and in practice, the algorithm performs much better. This is due to the two IF statements that limit the number of iterations that continue through to the inner loops. To get a better measure of the actual runtime cost, a different analysis is required.

3.3.2 Computational Cost

Given that the big-$\mathcal{O}$ analysis provides an asymptotic bound which does not always reflect the runtime performance, a different approach is required to measure and compare the computational efficiency of the PIPE algorithm. With such a measure, one can then effectively compare the efficiency of the algorithm after making changes and/or improvements. The term *Computational Cost* refers to the analysis of the algorithm based on counting the number of computationally expensive operations.

For the PIPE algorithm we have identified two operations that are performed continuously throughout the algorithm and are believed to consume the majority of the runtime. The first operation is the window comparison function which determines whether or not two windows are similar to one another. Not only is this operation performed repeatedly through the entire algorithm, it involves many lookups in the PAM120 matrix - an operation which, on its own, is considered expensive. The second operation is the determination of a protein's neighbours. Despite the fact that the protein-protein interaction information is efficiently stored as a graph, the determination of a protein's neighbours involves stepping through a linked list, resulting in repeated random memory accesses. This type of random memory access, often referred to as pointer chasing, cannot easily be predicted and makes it difficult to take advantage of the various memory hierarchies in a cache-based architecture. For both operations, counters were added to the program to measure the frequency of calls to each. The program was then run 1000 times on randomly chosen target protein pairs in order to come up with an average number of times each operation is executed in the comparison of two proteins. Table 4 presents the results of this analysis, including the average count, median, and standard deviation for both operations.

What is evident from Table 4 is that the number of window comparisons is far greater than the number of random memory accesses caused by pointer chasing. This provides fairly strong evidence that optimizing and/or reducing the number of

Operation	Average	Median	Std. Deviation
Window Comparison	5,114,082,336	2,744,671,790	7,356,690,492
Pointer Chasing	20,407	10,626	30,824

Table 4: Computational Cost Analysis of original PIPE program

window comparisons is key to improving the overall efficiency and performance of this algorithm. Also of note is that the standard deviation (measured in relation to the average) seems to be quite high. As one can see in Figures 9 and 10, the high standard deviation is due to the skewed nature of the histogram plots.

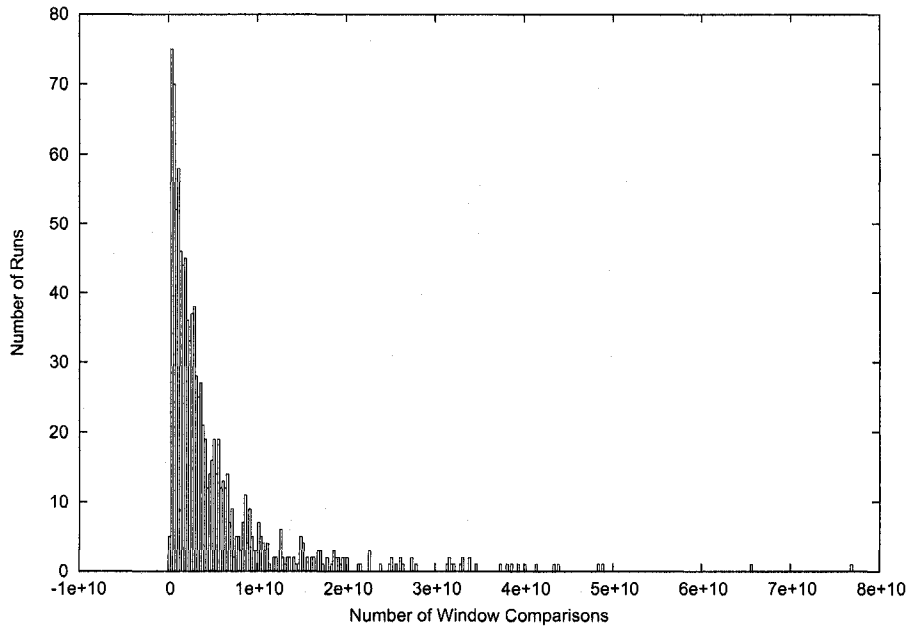

Figure 9: Window Comparisons Distribution (histogram of 1000 runs)

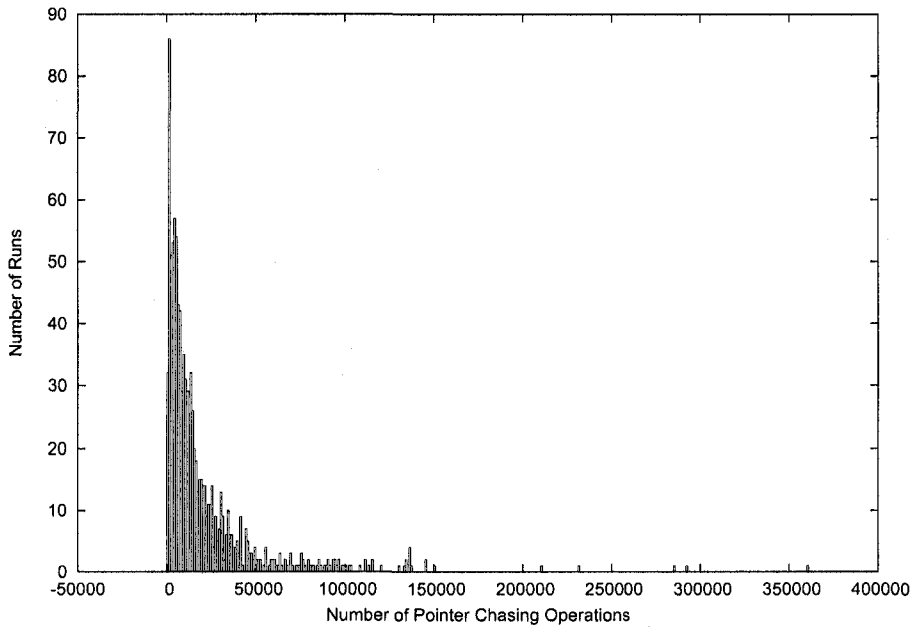

Figure 10: Pointer Chasing Distribution (histogram of 1000 runs)

Chapter 4

PIPE Improvements

The PIPE algorithm described in Chapter 3 was initially designed to predict the likelihood of interactions between protein pairs. It was parallelized using the MPI parallel programming paradigm. However, despite the use of 32 Intel Xeon processors, the average runtime for each protein pair made it infeasible to run the PIPE program against all 20 million protein pairs found in the yeast organism. Conservative estimates place the total runtime for all possible protein pairs at hundreds of years. Clearly, in order to exhaustively test all possible protein pairs, significant optimizations and/or algorithmic changes were required.

To achieve the necessary performance improvements in the PIPE algorithm the following process was followed. First, the PIPE implementation was optimized as

much as possible without changing the structure of the algorithm. As these optimizations did not result in a sufficient performance improvement, algorithmic changes were investigated.

In order to maximize the performance improvements of the original PIPE implementation, it was imperative to understand the behaviour of the original PIPE program. Behaviour refers to the overall flow of the application, including function call-graph and time spent in each subroutine. Program instrumentation was used for application profiling. Access to the processor's hardware counters was available through use of the Performance Application Programming Interface (PAPI) [9, 10], however, this level of analysis was deemed unnecessary.

4.1 Performance Optimizations

The first step in optimizing the PIPE program was to profile the application and determine the percentage of the runtime spent in each function. Function level profiling was done as the simple nature of the program's functions made it unnecessary to do loop and block level profiling. The profiling was performed on the single-processor implementation of the code in order to focus on the computational aspects of the program and not the inter-processor communication. Instrumentation information was collected after initial input of the protein sequence data and the interaction graph. In order to produce an accurate profile, the same set of optimization flags as used in the

original PIPE implementation were provided to the compiler with the exception that function inlining was not turned on. With function inlining turned on, performance improvements may be achieved, however, the application profile of this scenario does not correctly identify the time spent in each function. Table 5 provides the results of this initial profile.

Function Name	Percentage
PAM_index()	67.4%
compare_segment()	31.4%
find_B_in_C()	0.7%
find_Ai_in_C()	0.3%
All other functions	0.2%

Table 5: Profile of original PIPE program

What is clear from this initial profile is that the majority of the runtime is spent in the *PAM_index()* and *compare_segment()* functions. Together, these two functions are responsible for performing the window comparisons throughout the program and take up 98.8% of the runtime. The *PAM_index()* function is called by the function *compare_segment()* and is used when looking up corresponding amino acid pairs in the PAM120 matrix. It does this by searching through a character-to-index mapping array. On average, since there are a total of 23 different amino acid characters (20 standard amino acids plus 3 unresolved/unknown), there are 13.5 memory comparisons required before the correct index is found. This function is very inefficient and can be deleted all together if one represents the protein sequence using the PAM120

index values for each amino acid instead of their character representation.

This initial optimization was very easy to implement and results in a significant reduction in the number of memory accesses required each time the PAM120 matrix is applied to a pair of amino acids. The initial implementation requires, on average, $2 \times (1 + 23/2) + 1 = 26$ memory accesses to perform each PAM120 lookup (1 for each amino-acid character, 23/2 to convert each character to the PAM120 index value, and 1 for the final PAM120 lookup). The new implementation requires 3 memory accesses to perform this same operation (1 for each amino-acid character, and 1 for the final PAM120 lookup). Given that the *PAM_index()* function took 67.4% of the runtime in the original implementation and that this improvement requires approximately $1/10^{th}$ the number of memory accesses, it is estimated that this optimization results in a 3X - 4X performance improvement over the original program. In addition, it is believed that additional performance improvements may occur as a result of a more predictable memory access pattern within the *compare_segment()* function.

Following this initial improvement, a second application profile was performed. Table 6 provides the results of this profile.

This profile indicates that after the initial optimization, 65.3% of the runtime is still being spent in the *compare_segment()* function, however *find_B_in_C()* and *find_Ai_in_C()* now account for 34.5% of the runtime (up from 1%). Since these functions take up less than 35% of the runtime, no significant improvements were

Function Name	Percentage
compare_segment()	65.3%
find_B_in_C()	20.9%
find_Ai_in_C()	13.7%
All other functions	0.1%

Table 6: Profile of improved PIPE program

deemed possible through their optimization. Therefore, we return again to the *compare_segment()* function. This function is used to compare two windows and returns true or false whether or not they are similar to each other. In comparison to the original PIPE implementation, this function is now very basic and performs a minimal number of memory accesses, arithmetic operations, and conditional branches, therefore no significant optimizations were deemed possible. In order to achieve significant performance improvements a better algorithmic approach to the window comparisons was required.

Recall that the underlying algorithm compares every window of the first target protein, $A_i \in A$, to the windows of all other proteins. For each protein, C, that contains a window similar to A_i, the algorithm proceeds with the comparison of all windows of the second target protein, B, to the windows of the proteins that are known to interact with protein C. If one ignores the early break-out condition, upon completion of the algorithm, each A_i will have been compared with every window of every protein in $\mathcal{S}$. The way the algorithm is designed, each of these comparisons

is independent. If the cost of comparing two windows is σ, and there are n such windows in protein A and m such windows in protein X, the total cost of the window comparisons for A and X is $(\sigma \times n \times m)$.

What the PIPE algorithm does not take into consideration is the fact that sequential windows are simply a shift of one amino acid within the protein sequence. Put another way, if windows contain L amino acids, then A_i and A_{i+1} contain a common subsequence of $L-1$ amino acids. If we extend this to the window comparisons for proteins A and X, the comparison of A_i and X_k results in the same PAM120 lookups as the comparison of A_{i+1} and X_{k+1} with the exception of two amino acid pairs (the first and last of each window). Using this idea, if the cost of initially comparing two L-length windows, A_i and X_k, is σ, then one can effectively compare A_{i+1} and X_{k+1} with a cost of ϵ (two PAM120 lookups). The same can be said for A_{i+2} and X_{k+2}, and so on. Figure 11 demonstrates how this approach would be performed when comparing all window from protein A with all window from protein X, along with the associated cost of each comparison. If ϵ is smaller than σ, this approach will result in a speedup over the initial algorithm. In practice, σ is approximately 10 times greater than ϵ, therefore this approach to window comparisons was expected to result in an additional 10X speedup.

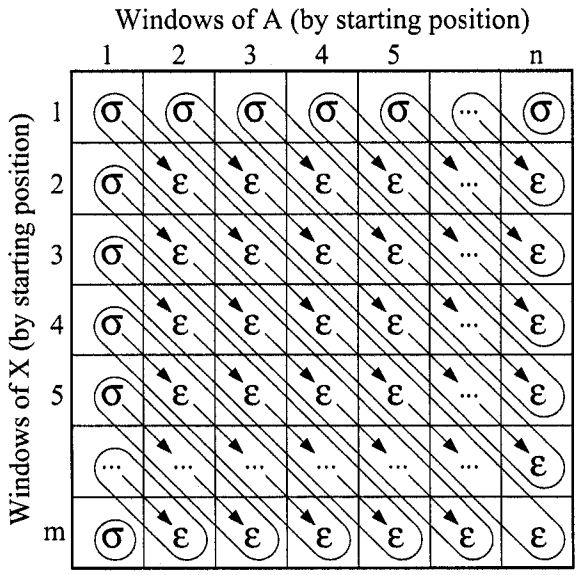

Figure 11: Window comparison costs using the diagonal sliding window approach

4.2 Algorithm Changes

4.2.1 Two Stage Approach

The second optimization suggested in Section 4.1 necessitates a structural change to the PIPE algorithm. Instead of comparing a single window from protein A to all other proteins, the algorithm compares all windows from protein A with all windows from a given protein X before comparing A with any other protein. This new approach requires splitting the PIPE program into two stages.

The first stage of the new algorithm implements the optimized window comparison approach and stores the pertinent information to disk. That is, for every window of every protein, determine the list of proteins that contain at least one window similar

to this window. To ensure that all possible window pairs are accounted for, this stage exhaustively compares all windows for all proteins with one each other. Only those windows that are similar to each other are stored to disk. Since the protein sequences themselves do not change, this process only needs to be performed once for the set of all proteins in an organism. Once this initial one-time work is done, subsequent PIPE runs can simply use the results without recomputing these window comparisons.

The amount of disk space required to store the results of these window comparisons is a function of the number of proteins, the number of windows per protein, and the number of similar windows found. For yeast, there are a total of 6304 proteins with approximately 455 windows per protein. On average, each window is similar to a window in 8.3 other proteins. The total disk space required to store the results of the window comparisons is approximately 65 MB.

The second stage implements the original PIPE algorithm without the window comparisons. For every window of the first target protein, extract from disk the proteins that contain similar windows. We then follow the original algorithm, and again extract similarity information from the neighbours of these proteins with windows of the second target protein.

The performance improvements obtained by splitting the algorithm into two stages are staggering. Not only does this version of the PIPE algorithm take advantage of both optimizations described in Section 4.1, it also eliminates the repetition of doing

the same window comparison multiple times. Section 4.3.2 describes the consequences of this in more detail. All combined, these changes result in a total speedup of approximately 16000X over the original PIPE implementation. This improvement has made it possible to run the PIPE algorithm against all possible protein pairs in yeast using the HPCVL's Linux Cluster.

To distinguish the improved PIPE algorithm from the original one, henceforth, we will refer to this new algorithm as PIPE-2.

4.2.2 Modified Parallel Work Distribution

Given that the PIPE-2 algorithm is now sufficiently fast to allow for the running of all possible 20 million protein pairs in yeast, one final change was made to the implementation to allow better workload distribution across all processors.

Recall that the original PIPE implementation parallelized the algorithm over windows of the first target protein. While this is still possible in the PIPE-2 implementation, the total runtime of <0.5 seconds per protein pair results in each processor performing a very small amount of work before broadcasting the results back to the master processor for final output. This is not an efficient approach to the the running of all possible protein pairs.

Instead, it was decided that each processor would perform an independent set

of protein-protein pair comparisons. This completely eliminates the need for inter-processor communication and makes it possible to run this algorithm on a set of distributed computers that do not have a parallel processing interface. Distributing the work in this manner provides more flexibility to the user, however, it can result in poor load balancing if the user is not careful. Since the runtime of a protein-protein interaction prediction is directly related to the number of windows in both target proteins, it is a bad idea to have a single processor perform all the predictions involving a single protein with a very long sequence length. Instead, the predictions for each protein should be equally distributed over all the processors. This is the approach that was implemented in PIPE-2. Another approach to resolve this load balancing issue is to use a master-slave approach to workload distribution. However, this adds additional complexity to the system and does not permit the running of the algorithm on a distributed system with no available parallel processing interface.

4.3 PIPE-2 Algorithm Cost

As with the original algorithm, the PIPE-2 algorithm was analyzed for both its theoretical (asymptotic) costs as well as its computational costs. However, unlike the original algorithm, which is designed to run against a single protein pair, the PIPE-2 algorithm is designed to run against a large number of protein pairs. For this reason, the algorithm first performs all possible window comparisons and stores the results

to disk. The algorithm then proceeds with the individual protein pair predictions using the results of this initial precomputation. For the cost analysis, we will provide the asymptotic and computational costs for the initial precomputation, the individual protein pair prediction, as well as all possible protein pair predictions. For each stage, comparisons will be made with the original PIPE algorithm.

4.3.1 Asymptotic Cost

Initial Precomputation

The initial precomputation exhaustively iterates over every pair of windows from every protein and determines whether the two windows are similar to one another. Figure 12 provides a high level overview of this algorithm along with the big-$\mathcal{O}$ cost of each step. For this analysis, there are two input variables. Let N be the number of windows in a given protein and let M be the total number of proteins.

For every protein $X \in \mathcal{S}$ {	$\mathcal{O}(M)$
For every protein $Y \in \mathcal{S}$ {	$\mathcal{O}(M)$
For every window $X_i \in X$ {	$\mathcal{O}(N)$
For every window $Y_j \in Y$ {	$\mathcal{O}(N)$
If $X_i \sim Y_j$ {	$\mathcal{O}(1)$
Write X_i and Y_j similarity information to disk	
} } } } }	

Figure 12: Big-$\mathcal{O}$ Analysis of the PIPE-2 window comparisons

This algorithm is somewhat inefficient given that it performs the comparison of

every pair of proteins twice. After comparing a protein X with another protein Y, it is not necessary to also compare the protein Y with the protein X. In practice, the algorithm does not repeat these redundant protein comparisons. This slight improvement reduces the number of protein comparisons by a factor of two, however, does not change the asymptotic cost. The overall asymptotic cost for the initial precomputation is therefore $\mathcal{O}(N^2 \times M^2)$. This initial precomputation stage is unique to the PIPE-2 algorithm and only needs to be performed once for the testing of all possible protein pairs in a given organism.

Individual Protein Pair Prediction

The individual protein pair prediction stage reads in the required window comparison information from disk, and then proceeds in the same way as the original PIPE algorithm. For this analysis, we must introduce a new variable, Q, which represents the number of proteins containing a window similar to a given window. Note that Q must be less than the total number of proteins in the organism, M. Figure 13 provides an algorithmic overview of this stage along with the big-$\mathcal{O}$ cost of each step.

The asymptotic cost for the PIPE-2 individual protein pair prediction is $\mathcal{O}(N^2 \times P \times Q)$. The cost of the original PIPE's individual protein pair prediction was $\mathcal{O}(N^2 \times P \times M)$. The asymptotic gain of the PIPE-2 algorithm when compared to the original PIPE algorithm is M/Q. For the yeast organism, there are a total of

For every window $A_i \in A$ {	$\mathcal{O}(N)$
For every protein $X \ni X_k$ such that $X_k \sim A_i$ {	$\mathcal{O}(Q)$
For every protein $Y \in \mathcal{S}$ such that $X \longleftrightarrow Y$ {	$\mathcal{O}(P)$
For every window $B_j \in B$ {	$\mathcal{O}(N)$
If $\exists\ Y_m \in Y$ such that $Y_m \sim B_j$ { $H[i][j] := H[i][j] + 1$	$\mathcal{O}(1)$
} } } } }	

Figure 13: Big-$\mathcal{O}$ Analysis of the PIPE-2 protein pair prediction stage

6304 proteins (M) and, on average, there are 8.3 proteins containing a window similar to a given window (Q). However, recall that the step with cost $\mathcal{O}(M)$ in the original PIPE implementation (see page 35) involved an *if()* statement which, in practice, would limit the cost of this step.

All Possible Protein Pair Predictions

In predicting all possible protein pair interactions for an organism, the PIPE-2 algorithm needs to perform the initial precomputation once, and then the individual protein pair prediction for all possible pairs. Again, let N be the number of windows in a given protein, let M be the total number of proteins, let P be the number of neighbours in the interaction graph for the given protein, and let Q be the number of proteins containing a window similar to a given window. Since there are a total of $(M \times (M-1))/2$ (or $\mathcal{O}(M^2)$) unique pairs, the asymptotic cost for predicting the interactions of all possible pairs is $\mathcal{O}((N^2 \times M^2) + (N^2 \times M^2 \times P \times Q)) =$

$\mathcal{O}(N^2 \times M^2 \times P \times Q)$. The equivalent asymptotic cost for the original PIPE algorithm is $\mathcal{O}(N^2 \times M^3 \times P)$. Again, as with the individual protein pair prediction, the asymptotic gain of the PIPE-2 algorithm when compared to the original PIPE algorithm is M/Q.

4.3.2 Computational Cost

As with the original PIPE program, an analysis of the PIPE-2 program's computational cost is done in order to measure the computationally expensive operations performed throughout the program. The two operations identified in Section 3.3.2 were window comparisons and pointer chasing. For the PIPE-2 algorithm, the pointer chasing operation is broadened to include all linked list traversals, and not just ones associated with searching the interaction graph.

Initial Precomputation

Table 7 provides the results of the application instrumentation analysis for the initial precomputation of all possible window comparisons. These numbers were obtained by adding counters throughout the code wherever these operations are performed.

Operation	Total Count
Window Comp.	4,105,863,304,922
Pointer Chasing	0

Table 7: Computational Cost Analysis of the PIPE-2 window comparisons

Since this initial precomputation stage only performs window comparisons, there are no costs associated with pointer chasing. This stage is unique to the PIPE-2 algorithm and therefore no comparison can be made to the original algorithm.

Individual Protein Pair Prediction

In order to allow meaningful comparison between the PIPE-2 algorithm and the original algorithm, the computational analysis for the individual protein pair prediction was done using the same 1000 randomly chosen pairs as in Section 3.3.2. For each randomly chosen pair, the number of window comparisons and pointer chasing operations was recorded. Table 8 provides the results of this analysis and includes, for comparison, the results from the original PIPE algorithm.

Algorithm	Operation	Average	Median	Std. Deviation
Original PIPE	Window Comp.	5,114,082,336	2,744,671,790	7,356,690,492
PIPE-2	Window Comp.	0	0	0
Original PIPE	Pointer Chasing	20,407	10,626	30,824
PIPE-2	Pointer Chasing	58,698,515	16,522,349	150,265,763

Table 8: Computational Cost Analysis of the PIPE-2 individual pair prediction

One thing to node in Table 8 is that the number of pointer chasing operations increases substantially from the original PIPE implementation to PIPE-2. This phenomenon is due to the fact that PIPE-2 stores the window comparison information (read in from disk) as linked lists and must traverse these lists throughout the algorithm. In the original PIPE implementation, the only pointer chasing operations

performed were those associated with searching the interaction graph for neighbours of a given protein.

All Possible Protein Pair Predictions

As was discussed in Section 4.3.1, predicting the interactions in all possible protein pairs for a given organism requires running the initial precomputation once, and then the individual pair prediction $(M \times (M-1))/2$ times. Table 9 provides the computational cost analysis for both the original PIPE and PIPE-2 programs. These values were obtained by scaling up the individual protein pair prediction results. For the PIPE-2 total costs, the analysis also includes the initial precomputation of all possible window comparisons.

Algorithm	Operation	Estimated Count
Original PIPE	Window Comp.	101,601,760,157,922,816
PIPE-2	Window Comp.	4,105,863,304,922
Original PIPE	Pointer Chasing	405,427,011,792
PIPE-2	Pointer Chasing	1,166,166,684,621,840

Table 9: Computational Cost Analysis for all-to-all protein pair predictions

What is clear from Table 9 is that in computing all possible protein-protein interactions, the PIPE-2 program reduces the number of window comparisons by a factor of approximately 25000X. This indicates that, when scaled up to all possible protein pair predictions, the original PIPE algorithm performs a large number of duplicate

comparisons. This is clearly not an efficient approach and explains the remarkable performance improvement obtained by PIPE-2.

With the decrease in window comparisons comes an approximate 3000X increase in the pointer chasing operations. This increase is due to the fact that, for each individual protein pair prediction, the window comparison information is read from disk and stored in memory as a series of linked lists. This tradeoff in window comparisons for pointer chasing operations is deemed acceptable given that a pointer chasing operation is much quicker than a window comparison, and given that the decrease in window comparisons (25000X) is much larger than the increase in pointer chasing operations (3000X).

Chapter 5

Results

5.1 All-to-All Performance Improvements

Chapter 4 described various optimizations and changes that were made to the original PIPE algorithm to improve its efficiency. The goal of these changes was to produce an improved implementation of the algorithm that would be capable of performing all possible protein-protein interaction predictions for the yeast organism.

The optimized PIPE algorithm, called PIPE-2, achieved its performance improvements through three main changes to the implementation. The first change removed the need for a function that converts amino acid characters to their corresponding PAM120 index values. This change was estimated to result in a 3X - 4X performance improvement with the possibility of additional improvements due to more predictable

memory access patterns. The second change took advantage of the fact that sequential windows only differ by the removal of one and the addition of one amino acid. The result is that following each full window comparison (which costs σ), subsequent window comparisons cost ϵ (i.e. $1/10^{th}$ the number of PAM120 lookups with $L = 20$). See Figure 11 on page 45 for a pictorial representation of this optimization. This change was estimated to result in a 10X improvement. The final change was an algorithmic modification which split the algorithm into two distinct stages. The first stage is the window comparison stage and is performed once for the entire organism. The second stage performs the individual protein pair prediction using the window comparison results stored on disk. This algorithmic change significantly lowers the number of repeated window comparisons performed throughout the running of all possible protein pairs.

Table 10 provides a statistical analysis of the runtime of 1000 randomly chosen protein pairs for the original PIPE implementation, the original implementation with the first optimization, and the final PIPE-2 implementation. The second optimization cannot be shown separate from the final PIPE-2 implementation since it necessitates the split-stage approach. For comparison, average speedup (over the original PIPE algorithm) is also provided. All numbers (with the exception of speedup) represent the runtime in seconds.

As with the computational analysis done in Chapters 3 and 4, what is interesting

Version	Average	Median	Std. Deviation	Speedup
Original PIPE	6,944.40	3,715.78	11,238.98	1 X
Original + 1^{st} Optimization	389.65	203.17	557.83	18 X
PIPE-2	0.43	0.14	1.51	16,150 X

Table 10: Runtime Analysis of 1000 randomly chosen protein pairs

in this table is that the standard deviation is quite high in comparison to the average runtime. Figures 14, 15 and 16 presents the runtime distribution histogram for each of these versions.

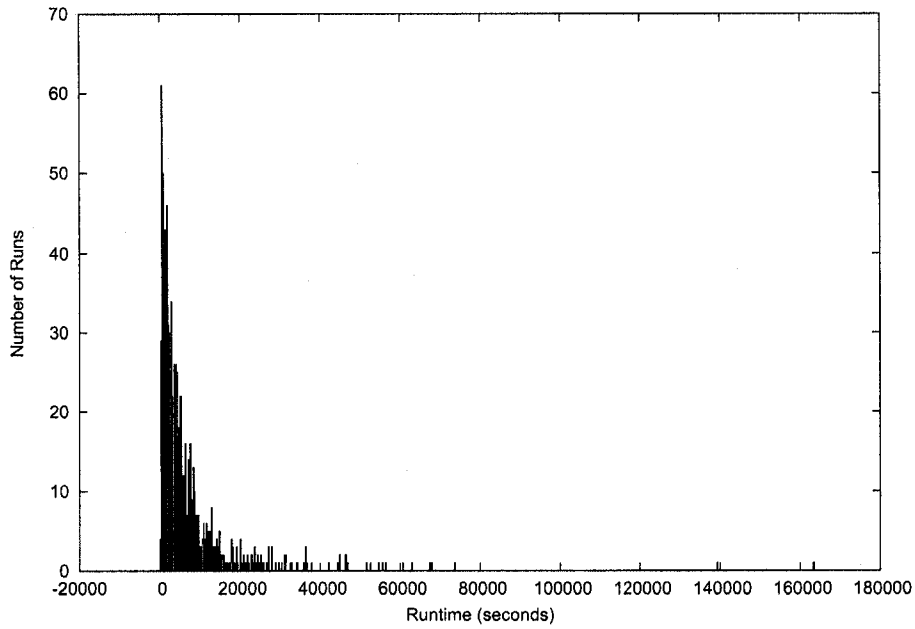

Figure 14: Original PIPE Runtime Distribution

The runtime distributions for Figures 14 and 15 are both skewed in the same manner as the window comparison and pointer chasing distributions from Section 3.3.2

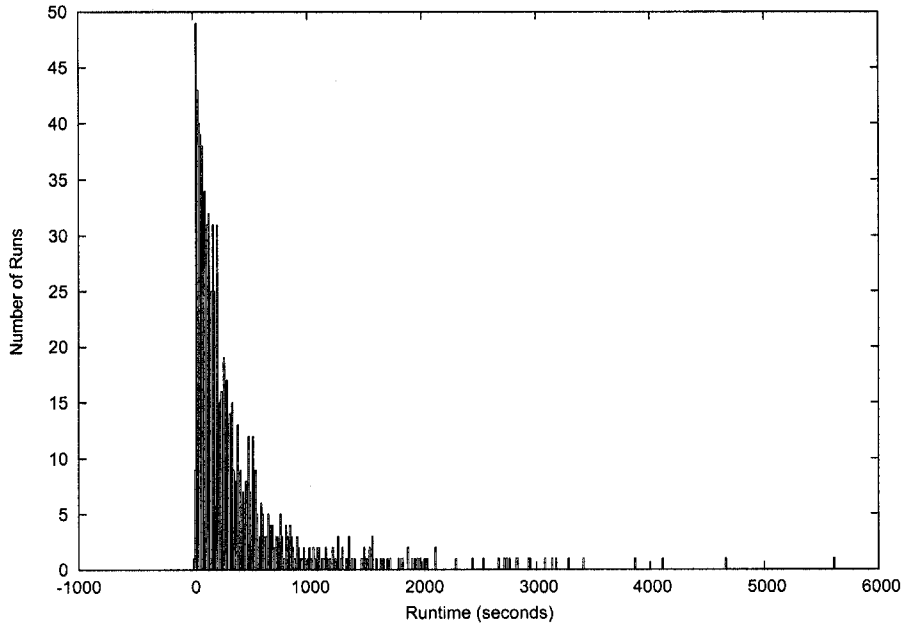

Figure 15: Original PIPE + 1^{st} Optimization Runtime Distribution

(pages 37 and 38). This confirms that the computational cost measured in Chapters 3 and 4 is a good gauge of the runtime performance. For the PIPE-2 algorithm (Figure 16), the runtime distribution seems to be more sharply skewed. This may be indicative of the new ratio between the time spent reading in the similarity information and interaction lists from disk and the time spent performing the actual protein-protein interaction prediction computation.

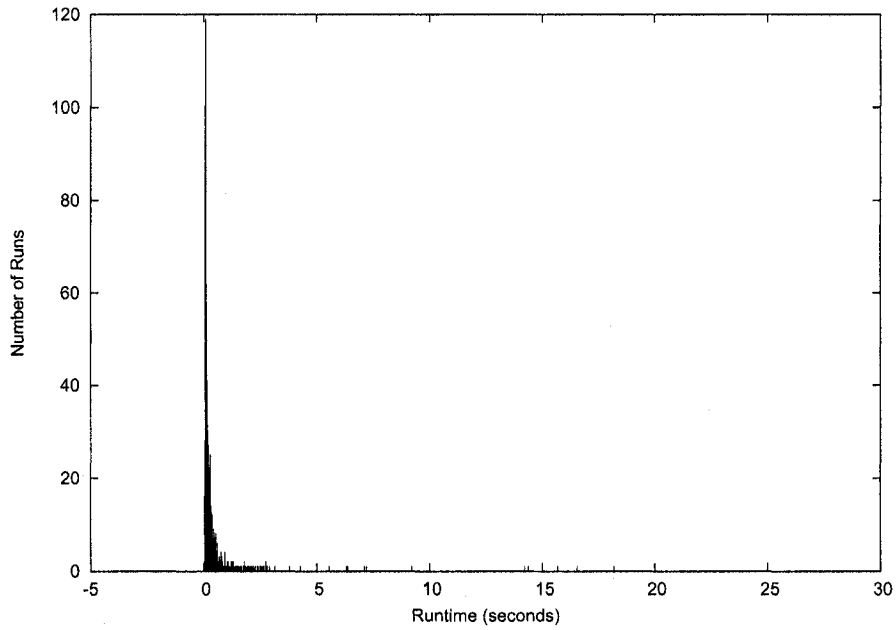

Figure 16: PIPE-2 Runtime Distribution

5.2 Impact of Results

The PIPE-2 implementation provides a 16 thousand times performance improvement over the original PIPE algorithm and implementation by Pitre et al. It is now possible to perform the all-to-all protein-protein interaction predictions in less than one week of total computation time on a 32-processor Linux Cluster. Results obtained from this experiment has lead to the re-evaluation of the effectiveness of the PIPE algorithm and has resulted in numerous parameter changes in order to increase the rate of true positives and decrease the rate of false positives. The changes we describe in Sections 5.2.2 and 5.2.3 were done primarily by Sylvain Pitre and Albert Chan under the direction of Frank Dehne and Ashkan Golshani and with assistance from other members of the Carleton University School of Computer Science Bioinformatics Group. These changes, and the corresponding advancements would not have been possible without the performance improvements provided by the PIPE-2 algorithm and implementation.

5.2.1 Maximum Peak

The original PIPE algorithm determines whether or not two proteins interact by first producing a two-dimensional matrix whose entries represent the number of times corresponding windows are similar to windows in protein pairs that are known to interact. The higher the value, the more likely it is that these windows play a role

in the biological interaction of the two proteins, and the more likely it is that the two target proteins also interact. Determining an interaction in the original PIPE algorithm was simply a matter of evaluating the maximum value of the resulting two-dimensional matrix, and determining if it surpassed a pre-determined threshold.

In the original PIPE implementation, the determination of the maximum peak threshold was made after several hundred trial runs of protein pairs that were known to interact and pairs that were known not to interact. What was missing was large random sampling of the population to determine the effectiveness of the chosen maximum peak threshold value at filtering out randomly chosen (i.e. non-interacting) sequences. With the drastic performance improvements of PIPE-2, it is now possible to conduct such a large scale survey.

The large scale survey proceeded as follows. First, 20000 protein sequences were randomly generated to form 10000 random protein pairs. Each sequence was comprised of 500 amino acids (to approximate the average protein sequence length in yeast). The random protein sequences were generated by combining randomly selected sequences of five amino acids from the protein sequence database. This was done in order to maintain the amino acid frequency distribution as well as account for the fact that the occurrence of a given amino acid is often dependent on surrounding amino acids. Once the 10000 protein pairs were generated, the PIPE-2 algorithm was run against these pairs and the maximum peak value for each pair was recorded.

Figure 17 presents the cumulative probability results for this experiment.

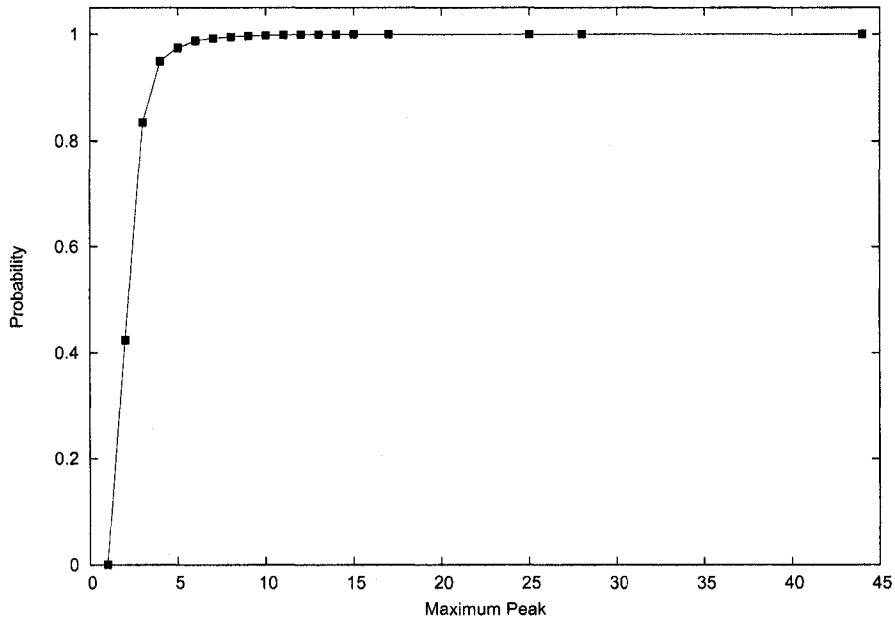

Figure 17: Maximum Peak Cumulative Probability

This figure tells us that 99.8% of randomly generated protein pairs will have a maximum peak value less than 10. Since it is assumed that each of these randomly generated sequences do not actually correspond to interacting protein pairs, using a maximum peak threshold of 10 should result in a 0.2% rate of false positives. This was deemed to be an acceptable rate of false positives and confirmed the original PIPE algorithm's choice of using 10 as the maximum threshold.

When the all-to-all PIPE-2 experiment was run, we were surprised by the number of protein interactions identified. We were expecting approximately 16000 - 26000 true positives and approximately 40000 false positives (0.2% of 20 million) for a total

of 56000 - 66000 interactions. Instead, PIPE-2 reported over 2.8 million interactions. Since the number of true positives is not likely to be greater than 30000 [17], the majority of these 2.8 million reported interactions are false positives. A possible explanation for this greater than expected rate of false positives is the low-complexity areas of protein sequences. Choosing a higher maximum peak threshold would reduce the number of false positives, but it would also reduce the number of true positives. This lead to the search for alternate methods for determining protein-protein interactions from the two-dimensional matrix generated by PIPE-2.

5.2.2 Average and Variance

In order to determine a better measure for analyzing the protein-protein interaction results produced by PIPE-2, the statistical package SAS was used to generate a best-fit classification tree for a given set of parameters. The input to the SAS routine included results from a set of known true positives and true negatives. The parameters considered when building the classification tree included maximum peak, average height, variance, as well as other custom parameters.

The resulting classification tree indicated that the average height and variance were the best parameters for differentiating true positives from true negatives. This tree classifies protein pairs with average height ≥ 1.0 and variance ≤ 2.9 as interacting pairs. Otherwise, the protein pair is classified as non-interacting.

Given this new selection criterion for interacting pairs, the all-to-all PIPE-2 experiment was run a second time. Once complete, an analysis was performed to determine the effectiveness of the new selection criterion. In particular, sensitivity (rate at which we can detect true positives) and specificity (rate at which we can detect true negatives) were measured. To perform this analysis, it was necessary that we have a list of true positives and true negatives in order to measure the effectiveness of the algorithm. A list of 1000 true positives was constructed by taking the intersection of several published interaction databases. This was done to increase the level of confidence in the list of true positives. The list of plausible negatives was constructed randomly, in the manner described in section 5.2.1.

The results of this analysis were very positive. With an interacting pair identified by average height ≥ 1.0 and variance ≤ 2.9, PIPE-2 was able to correctly identify 93.00% of the true positives and 99.95% of the "true" negatives. Extrapolating to the entire all-to-all experiment, we would expect to see approximately 19500 true positives (93% of the expected 30000 total positives) and 10400 false positives (0.05% of the expected 20 million negatives). Therefore, we expect PIPE-2 to identify 29900 positive interactions. The actual number of interactions identified by PIPE-2 for the all-to-all experiment using this selection criterion was 21631. This is 72.3% of the interactions expected. Section 5.2.3 will outline a possible reason for this discrepancy and discuss other problems with this selection criterion.

5.2.3 PIPE's Prediction Capabilities

The results from the all-to-all PIPE-2 experiment appear to be very promising. With an estimated sensitivity of 93.00% and a specificity of 99.95%, the PIPE-2 algorithm seems to produce results that are significantly better than other experimental and computational methods. However, this is somewhat misleading. When we report that PIPE-2 is able to predict 21631 interactions, this includes those interactions that were known in advance and provided to PIPE-2 in the form of the interaction graph. For the all-to-all experiment, the interaction graph consisted of 15118 known interactions. These interactions were compiled by combining various publicly available databases. In fact, of the 15118 known interactions provided to PIPE-2, only 14354 (94.95%) of these interactions were actually predicted by PIPE-2. That is, PIPE-2 is not able to identify all previously known interactions. Furthermore, PIPE-2 is only actually predicting 7277 new interactions. Without secondary confirmation through laboratory experiments, it is impossible to say what percentage of these new predicted interactions are true interactions versus false positives. This raises a fundamental question about PIPE's ability to predict previously unknown interactions.

To answer this question, a series of cross validation tests were designed to test PIPE's ability to predict interactions beyond those provided in the interaction graph. The first test was a 10-fold cross validation that involved randomly removing 10% of the known interaction graph and then re-running PIPE-2 to see what percentage

of these interactions were predicted by PIPE-2. After running 10 such tests, the resulting prediction rates are averaged together to produce an overall prediction rate for PIPE-2. The results of this experiment were discouraging. PIPE-2 was able to predict less than 5% of the removed interactions. Clearly either something was wrong with the way we were approaching these cross validation tests or there is a fundamental limitation to PIPE's ability to predict protein-protein interactions.

In a attempt to isolate the cause of the problem, the 10-fold cross validation test was repeated, however, this time we used an interaction graph in which we had high confidence instead of the one loosely compiled from publicly available databases. The high confidence interaction database consisted of the same known interactions described in Section 5.2.2. This test was done to determine if the cause of the problem might be the high error rate in the known interaction list that is provided to PIPE-2. By using a high confidence interaction database, we were hoping to eliminate most of the "noise" attributed to errors in the interaction graph. Unfortunately, the results of this test were only moderately better than the initial cross validation test, providing a sensitivity of just over 5% for the removed interactions.

In a final attempt to identify the cause of our problem, a third cross validation test was conducted. This test used the same high confidence interaction graph, however, instead of removing 10% of the interactions, only one interaction was removed. This test was repeated numerous times removing a different interaction each time.

Analyzing the resulting 2-dimensional matrices quickly identified what was causing the problem.

The reason PIPE-2 was not able to predict much beyond the provided interaction graph is due to the new threshold parameters used. In particular, using an average value ≥ 1.0 filters out most interactions that are not provided in the interaction graph. The reason for this is as follows. When PIPE-2 is predicting a pair that is already in the interaction graph, every entry of the resulting two-dimensional matrix is guaranteed to have a value ≥ 1.0 and the average height is therefore also ≥ 1.0. If we remove this interaction from the graph, the average height drops by 1.0 and unless this protein pair had a maximum value of ≥ 2.0 it is no longer identified by PIPE-2 as an interaction.

The implication of this observation is twofold. First, average height should not be used as a threshold parameter since it is greatly influenced by the occurrence of a protein pair within the known interaction graph. Second, when generating the classification tree, it is best to use a set of true positives that is different from those provided to PIPE-2 in the interaction graph.

Chapter 6

Future Work

The work presented in this thesis has expanded our understanding of the PIPE algorithm and provides insights into areas that require further exploration and analysis. Some of these areas are currently being pursued by the Carleton University School of Computer Science Bioinformatics group while others are left as future research projects.

6.1 Parameter Tuning

In Section 5.2.1 we discussed the results obtained from PIPE-2 when using maximum peak as a selection criterion for determining protein interactions. Despite having a specificity and sensitivity rate comparable to most commonly used experimental methods [29], this selection criterion produces millions of false positives when run on

all possible protein pairs. This makes it nearly impossible to discern between true positives and false positives.

In Section 5.2.2 we examined classification parameters which were predicted by a best-fit classification tree to be a useful selection criterion. This lead to the realization that selection criterion may be influenced by the occurrence of a protein pair within the known interaction graph. Therefore, in order to effectively test the prediction capability of PIPE, it is important to remove the set of true positives from the interaction graph when testing PIPE's prediction capability.

It is clear from the all-to-all results that better classification parameters are required for PIPE to correctly predict previously unknown protein-protein interactions. It has been suggested that we return to the original approach which identifies interactions based on the occurrence of peaks in the three-dimensional histogram. However, we should develop a more sophisticated peak detection algorithm that will look for nicely shaped "mountainous" peaks as opposed to random spikes which may be a result of errors in the known interaction graph or an impact of low-complexity regions within protein sequences.

6.2 Low-Complexity Filtration

The PIPE algorithm relies on the hypothesis that there are a finite number of polypeptide sequences that are responsible for mediating the interaction between protein

pairs. The PIPE algorithm predicts the likelihood of an interaction between two proteins by measuring the number of co-occurrences of fixed-length polypeptides with protein pairs that are known to interact. What the PIPE algorithm does not take into consideration is the impact that low-complexity regions may have on the three-dimensional histogram representing window similarity with known interacting pairs. Low-complexity regions are subsequences of amino acids of biased composition. These regions have been shown to produce many false positives in pairwise alignments and the removal of such regions has been shown to significantly improve the reliability of sequence similarity searches [35].

A natural extension to the PIPE algorithm would be to perform low-complexity filtration on the amino acid sequences to eliminate the impact of these regions on the resulting data. This filtration could either be performed during the initial window comparison stage or during the prediction stage of the PIPE-2 algorithm. It is unclear how the removal of these regions would impact the resulting data, however it is likely to decrease the number of false positives for the window comparisons thereby increasing our overall confidence in the results produced.

6.3 Human Protein-Protein Interactions

Assuming the PIPE algorithm is improved sufficiently to provide consistent prediction of new protein-protein interactions, and that the number of false positives can

be reduced to within the same order of magnitude as the number of true positives, it may be worth considering its use in exploring more complex organisms. The human genome is obviously of great interest and, given its size, would pose significant computational challenges for PIPE-2. The runtime for the two stages of PIPE-2 are influenced by several structural features of the organism and are therefore analyzed separately.

6.3.1 Stage 1 - Window Comparisons

The window comparisons stage compares all possible windows of all proteins with one another and stores the relevant information to disk. The asymptotic cost for this stage is $\mathcal{O}(N^2 \times M^2)$, where N is the number of windows in each protein sequence and M is the total number of proteins. In practice, when performing the window comparisons stage, we don't compare a protein with itself and we don't compare a protein pair twice (that is, we compare $\mathcal{A}$ with $\mathcal{B}$ but not $\mathcal{B}$ with $\mathcal{A}$). Therefore, the total number of window comparisons is $(N^2 \times M \times (M-1))/2$.

Yeast has a total of 6304 proteins with an average of 455 windows per protein (with a window length $L = 20$). The total number of window comparisons is therefore $(455^2 \times 6304 \times 6303)/2 = 4.1 \times 10^{12}$.

Humans, on the other hand, have between 20000 - 30000 proteins [7, 31] with an average of 491 windows per protein (with a window length $L = 20$) [34]. If we

assume there are a total of 30000 proteins, the total number of window comparisons is $(491^2 \times 30000 \times 29999)/2 = 108.5 \times 10^{12}$.

Given that this stage consists only of performing all possible window comparisons, the total runtime should scale linearly with the number of such operations. Therefore, the estimated runtime for the precomputation stage for the human genome is 26 times that of yeast. Given that the yeast precomputation took less than 30 minutes using 32 processors, the precomputation for humans should take less than 13 hours using the same parallel computer.

6.3.2 Stage 2 - Protein Pair Prediction

The protein pair prediction stage performs the individual predictions for all possible protein pairs in the organism. The asymptotic cost for each individual protein pair prediction is $\mathcal{O}(N^2 \times P \times Q)$, where N is the number of windows in the protein sequence, P is the number of neighbours of a given protein in the interaction graph, and Q is the number of proteins that have a window similar to a given window. In the PIPE-2 algorithm, the only significant computation is performed within the inner most loop, therefore the asymptotic cost accurately reflects the amount of work to be performed. The runtime of this stage should therefore scale linearly with its asymptotic cost.

For the yeast organism, there are an average of 455 windows per protein sequence.

The interaction graph used throughout the PIPE experiments consisted of 15118 known interactions involving 4716 distinct proteins. This works out to 6.4 interactions per protein. On average, the number of proteins containing at least one window similar to a given window is 8.3. Therefore, the cost of an individual protein pair prediction is $455^2 \times 6.4 \times 8.3 = 11.0 \times 10^6$. When multiplied by the total number of possible protein pairs $((M \times (M-1))/2)$ this works out to an all-to-all prediction cost of 218.4×10^{12}.

Humans, on the other hand, have an average of 491 windows per protein sequence. Results from a variety of methods has identified 31609 protein-protein interactions involving 7748 proteins [33]. This works out to an average of 8.2 interactions per protein. No data exists on the average number of proteins containing a window similar to a given window, however, if we extrapolate from the yeast organism (assuming it scales linearly with total number of windows) we arrive at an average of $8.3 \times (30000 \times 491)/(6304 \times 455) = 42.6$ proteins containing a window similar to a given window. Therefore, the cost of an individual protein pair prediction is $491^2 \times 8.2 \times 42.6 = 84.2 \times 10^6$. When multiplied by the total number of possible protein pairs this works out to an all-to-all prediction cost of 37.9×10^{15}.

The all-to-all protein pair prediction stage for the human genome is approximately 173 times more costly than it is for yeast. Given that this stage took approximately 5 days to complete on a cluster of 32 processors, performing this stage for human

would take approximately 865 days on this same system. Assuming the availability of a modern HPC system with 512 processors (with each processor twice as fast as those currently available to us), the computation would take approximately 27 days to complete. This is by no means an infeasible task, however, before this is to be considered, significant improvements to the PIPE algorithm are required to better distinguish true positives from false positives.

Chapter 7

Conclusions

This thesis presented and discussed improvements made to the PIPE algorithm - an algorithm used to identify protein-protein interactions in the yeast organism. These improvements focused on improving the efficiency of the fixed-length polypeptide comparisons and reducing the total number of such comparisons required. The combined improvements resulted in a 16 thousand times performance improvement over the original implementation of the algorithm.

The dramatic performance improvement achieved by these changes has made it possible to run PIPE-2 on all possible yeast protein pairs. The result has been a more complete understanding of the PIPE algorithm, including the discovery of areas for improvements and shortcomings not previously known. Assuming these shortcomings can be addressed, the new optimized PIPE algorithm, PIPE-2, is predicted to be able

to tackle larger organisms such as the laboratory mouse and the human. By improving our understanding of the cellular functions present in all organisms we are moving closer to deciphering their internal workings; paving the way for future biological discoveries and breakthroughs.

Bibliography

[1] S. Altschul. Amino Acid Substitution Matrices from an Information Theoretic Perspective. *Journal of Molecular Biology*, 219(3):555–565, 1991.

[2] A. Archakov, V. Govorun, A. Dubanov, Y. Ivanov, A. Veselovsky, et al. Protein-protein interactions as a target for drugs in proteomics. *Proteomics*, 3(4):380–391, 2003.

[3] A. Aytuna, A. Gursoy, and O. Keskin. Prediction of protein-protein interactions by combining structure and sequence conservation in protein interfaces. *Bioinformatics*, 21(12):2850–2855, 2005.

[4] D. Burger, J. Goodman, and A. Kagi. Memory bandwidth limitations of future microprocessors. In *Proceedings of the International Conference on Computer Architectures (ICCA)*, pages 78–89, 1996.

[5] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, et al. An evaluation of global address space languages: co-array fortran and unified

parallel c. In *Proceedings of the symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 36–47, 2005.

[6] J. Cohen. Bioinformatics - An Introduction for Computer Scientists. *ACM Comput. Surv.*, 36(2):122–158, 2004.

[7] International Human Genome Sequencing Consortium. Finishing the euchromatic sequence of the human genome. *Nature*, 431(7011):931–945, 2004.

[8] M. Dagenais, K. Yaghmour, C. Levert, and M. Pourzandi. *Software Performance Analysis.* Unpublished Manuscript, 2005.

[9] J. Dongarra, K. London, S. Moore, P. Mucci, and D. Terpstra. Using PAPI for hardware performance monitoring on Linux systems. In *Proceedings of Linux Clusters: The HPC Revolution*, 2001.

[10] J. Dongarra, S. Moore, P. Mucci, K. Seymour, and H. You. Accurate Cache and TLB Characterization Using hardware Counters. In *Proceedings of the International Conference on Computational Science (ICCS)*, 2004.

[11] K. Dowd and C. Severance. *High Performance Computing.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1998.

[12] R. Edgar, M. Domrachev, and A. Lash. Gene Expression Omnibus: NCBI gene expression and hybridization array data repository. *Nucleic Acids Research*, 30(1):207–210, 2002.

[13] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared Memory Programming.* John Wiley & Sons, Inc., Hoboken, NJ, USA, 2005.

[14] M. Feldman. Is Supercomputing Going Hetero? Technical report, September 2006. HPC Wire, http://www.hpcwire.com/hpc/897414.html.

[15] S. Fields and O. Song. A novel genetic system to detect protein-protein interactions. *Nature*, 340(6230):245–246, 1989.

[16] N. Galoppo, N. Govindaraju, M. Henson, and D. Manocha. LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware. In *Proceedings of the ACM/IEEE Supercomputing Conference*, pages 3–3, 2005.

[17] A. Grigoriev. On the number of protein-protein interactions in the yeast proteome. *Nucleic Acids Research*, 31(14):4157–4161, 2003.

[18] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, et al. *MPI — The Complete Reference: Volume 2, The MPI Extensions.* MIT Press, Cambridge, MA, USA, 1998.

[19] C. Harbison, D. Gordon, T. Lee, N. Rinaldi, K. Macisaac, et al. Transcriptional regulatory code of a eukaryotic genome. *Nature*, 431(7004):99–104, 2004.

[20] Y. Ho, A. Gruhler, A. Heilbut, G. Bader, L. Moore, et al. Systematic identification of protein complexes in saccharomyces cerevisiae by mass spectrometry. *Nature*, 415(6868):180–183, 2002.

[21] M. Huerta, F. Haseltine, Y. Liu, G. Downing, and B. Seto. NIH Working Definition of Bioinformatics and Computational Biology. Technical report, 2000.

[22] R. Jansen, H. Yu, D. Greenbaum, Y. Kluger, N. Krogan, et al. A Bayesian networks approach for predicting protein-protein interactions from genomic data. *Science*, 302(5644):449–453, 2003.

[23] N. Krogan, G. Cagney, H. Yu, G. Zhong, X. Guo, et al. Global landscape of protein complexes in the yeast Saccharomyces cerevisia. *Nature*, 440(7084):637–643, 2006.

[24] Z. Ling and V. Prasanna. High Performance Linear Algebra Operations on Reconfigurable Systems. In *Proceedings of the ACM/IEEE Supercomputing Conference*, pages 2–2, 2005.

[25] E. Marcotte, M. Pellegrini, H. Ng, D. Rice, T. Yeates, and D. Eisenberg. Detecting Protein Function and Protein-Protein Interactions from Genome Sequences. *Science*, 285:751–753, 1999.

[26] E. Nabieva, K. Jim, A. Agarwal, B. Chazell, and M. Singh. Whole-proteome prediction of protein function via graph-theoretic analysis of interaction maps. *Bioinformatics*, 21(Suppl.):i302–i310, 2005.

[27] U. Ogmen, O. Keskin, A. Aytuna, R. Nussinov, and A. Gursoy. PRISM: protein interactions by structural matching. *Nucleic Acids Research*, 33(Web Server Issue):W331–W336, 2005.

[28] F. Pazos and A. Valencia. Similarity of phylogenetic trees as indicator of protein-protein interaction. *Protein Engineering*, 14(9):609–614, 2001.

[29] S. Pitre, F. Dehne, A. Chan, J. Cheetham, A. Golshani, et al. PIPE: a protein-protein interaction prediction engine based on the re-occurring short polypeptide sequences between known interacting protein pairs. *BMC Bioinformatics*, 7(365), 2006.

[30] Human Genome Program. Genomics and Its Impact on Science and Society: A 2003 Primer. 2003.

[31] Human Genome Program. Functional and Comparative Genomics Fact Sheet. Technical report, 2005.

[32] M. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, New York, NY, USA, 2004.

[33] A. Ramani, R. Bunescu, R. Mooney, and E. Marcotte. Consolidating the set of known human protein-protein interactions in preparation for large-scale mapping of the human interactome. *Genome Biology*, 6:r40, 2005.

[34] M. Sakharkar, P. Kangueane, K. Sakharkar, and Z. Zhong. Huge proteins in the human proteome and their participation in hereditary diseases. *In Silico Biology*, 6(4):0026, 2006.

[35] H. Shin and S. Kim. A new algorithm for detecting low-complexity regions in protein sequences. *Bioinformatics*, 21(2):160–170, 2005.

[36] J. Sloan. *High Performance Linux Clusters with OSCAR, Rocks, OpenMosix, and MPI.* O'Reilly Media, Inc., Sebastopol, CA, USA, 2004.

[37] M. Snir and S. Otto. *MPI — The Complete Reference: The MPI Core.* MIT Press, Cambridge, MA, USA, 1998.

[38] A. Tong, G. Lesage, G. Bader, H. Ding, H. Xu, et al. Global Mapping of the Yeast Genetic Interaction Network. *Science*, 303:808–813, 2004.

[39] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The Potential of the Cell Processor for Scientific Computing. In *Proceedings of the Computing Frontiers Conference*, 2006.

[40] S. Wong, L. Zhang, A. Tong, Z. Li, D. Goldberg, et al. Combining biological networks to predict genetic interactions. *PNAS*, 101(44):15682–15687, 2004.

[41] W. Wulf and S. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, 1995.

[42] H. Yu, A. Paccanaro, V. Trifonov, and M. Gerstein. Predicting interactions in protein networks by completing defective cliques. *Bioinformatics*, 22(7):823–829, 2006.

[43] H. Zhu, M. Bilgin, R. Bangham, D. Hall, A. Casamayor, et al. Global Analysis of Protein Activities Using Proteome Chips. *Science*, 293(5537):2101–2105, 2001.