Analysis and Abstraction of Parallel Sequence Search

Christopher Joseph Goddard

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Wu-Chun Feng, Chairman
Godmar Back, Committee Member
Eli Tilevich, Committee Member

September 5, 2007
Blacksburg, Virginia

Keywords: sequence search, BLAST, parallelism, grid framework
Copyright 2007, Christopher Joseph Goddard

Analysis and Abstraction of Parallel Sequence Search

Christopher Joseph Goddard

(ABSTRACT)

The ability to compare two biological sequences is extremely valuable, as matches can suggest evolutionary origins of genes or the purposes of particular amino acids. Results of such comparisons can be used in the creation of drugs, can help combat newly discovered viruses, or can assist in treating diseases.

Unfortunately, the rate of sequence acquisition is outpacing our ability to compute on these data. Further, traditional dynamic programming algorithms are too slow to meet the needs of biologists, who wish to compare millions of sequences daily. While heuristic algorithms improve upon the performance of these dated applications, they still cannot keep up with the steadily expanding search space.

Parallel sequence search implementations were developed to address this issue. By partitioning databases into work units for distributed computation, applications like mpiBLAST are able to achieve super-linear speedup over their sequential counterparts. However, such implementations are limited to clusters and require significant effort to work in a grid environment. Further, their parallelization strategies are typically specific to the target sequence search, so future applications require a reimplementation if they wish to run in parallel.

This thesis analyzes the performance of two versions of mpiBLAST, noting trends as well as differences between them. Results suggest that these embarrassingly parallel applications are dominated by the time required to search vast amounts of data, and not by the communication necessary to support such searches. Consequently, a framework named gridRuby is introduced which alleviates two main issues with current parallel sequence search applications; namely, the requirement of a tightly knit computing environment and the specific, hand-crafted nature of parallelization. Results show that gridRuby can parallelize an application across a set of machines through minimal implementation effort, and can still exhibit super-linear speedup.

Dedication

I dedicate this thesis to my family, as they have been with me through so much. I would like to thank my mom, dad, sister, and grandma for their support (both emotional and financial) and for tolerating me when I visited home. I would like to thank my fiancée Vy for her unconditional support, patience, and love; without her, I would be lost.

I love you all.

Acknowledgements

This thesis could not have been completed without the assistance of some particularly helpful people. I would like to thank:

- My advisor, Dr. Wu Feng, for introducing me to the world of bioinformatics and for the time he gave me in his rather busy life.
- Dr. Godmar Back for his excellent instruction in operating systems and networking, and for suggesting that I should "look at the source."
- Dr. Eli Tilevich for the motivation to produce a "cool project" and for tolerating my desire to play with the Ruby language.
- Nathan Baker for his tremendous effort toward gridRuby and for the time he devoted to helping us relax through video games.
- The folks of ARC@VT, specifically Luke Scharf, for access to System X and for help when I ran into problems.
- Dr. Cal Ribbens, Dr. Mark Gardner, Jeremy Archuleta, and Rachel Hall Smith for their help and guidance throughout.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Goals 2

2 Sequence Searching with a Single Processor **4**

2.1 Basic Sequence Alignment 4

2.2 Smith-Waterman and Needleman-Wunsch 5

2.3 Heuristic Approaches 6

2.3.1 FASTP and FASTA 6

2.3.2 BLAST 7

2.4 Other Work 8

3 Sequence Search Parallelization in Tightly-Coupled Environments **9**

3.1 Introduction 9

3.2 Conventions 10

3.3 mpiBLAST 10

3.3.1 mpiBLAST-1.2 12

3.3.2 pioBLAST 13

3.3.3 mpiBLAST-1.4 16

3.3.4 mpiBLAST-PIO 17
3.4 Other Work 19
4 Analysis of Parallel Sequence Search in Tightly-Coupled Environments 20
4.1 Introduction 20
4.2 Conventions 20
4.3 Experimentation Variables 21
4.3.1 Hardware 21
4.3.2 Data 22
4.3.3 Toolkit 22
4.3.4 Method of Profiling 23
4.4 mpiBLAST-1.2 Analysis 23
4.4.1 High-level Discussion 24
4.4.2 Discussion of Non-search Time and Overhead 28
4.5 mpiBLAST-PIO Analysis 29
4.5.1 High-level Discussion 30
4.5.2 Discussion of Non-search Time and Overhead 33
4.6 Version Comparison 36
4.7 Potential Improvements 40
4.7.1 Parallelized Collection of Database Statistics 40
4.7.2 Task Threading at the Master 41
4.7.3 Adaptive Decomposition 42
4.8 Conclusion 42
5 Supporting Parallel Sequence Searches in Heterogeneous Environments 44

5.1 Introduction 44
5.2 Motivation 45
5.3 Conventions 46
5.4 The Framework 47
5.4.1 Justification of Ruby Language 48
5.4.2 Node Types 49
5.4.3 Integration 52
5.5 An Example: BLAST Integration 53
5.5.1 Configuration 53
5.5.2 Adhesion 58
5.5.3 Execution 58
5.6 Results 60
5.7 Related Work 61
5.8 Conclusion 63

6 Conclusion **64**

6.1 Future Work 64
6.2 Reflection 66
6.3 Conclusion 67

Bibliography **69**

List of Figures

3.1 Typical database and query segmentation patterns. 11

3.2 mpiBLAST-0.9 and mpiBLAST-1.2 output processes. 13

3.3 mpiBLAST-1.2 and pioBLAST output processes. 15

3.4 mpiBLAST-PIO output process. 18

4.1 Speedup of mpiBLAST-1.2. 24

4.2 Search time of mpiBLAST-1.2. 25

4.3 Efficiency of mpiBLAST-1.2. 26

4.4 Efficiency of the BLAST search in mpiBLAST-1.2. 27

4.5 Search related and non-search related time of mpiBLAST-1.2. 27

4.6 Breakdown of non-search related time in mpiBLAST-1.2. 28

4.7 Speedup of mpiBLAST-PIO. 30

4.8 Search time of mpiBLAST-PIO. 31

4.9 Efficiency of mpiBLAST-PIO. 32

4.10 Efficiency of the BLAST search in mpiBLAST-PIO. 32

4.11 Search related and non-search related time of mpiBLAST-PIO. 33

4.12 Breakdown of non-search related time in mpiBLAST-PIO. 34

4.13 Speedup comparison of mpiBLAST-1.2 and mpiBLAST-PIO. 37

4.14 Efficiency comparison of mpiBLAST-1.2 and mpiBLAST-PIO. 38

4.15 Comparison of non-search related time in mpiBLAST-1.2 and mpiBLAST-PIO. 38

5.1 A YAML configuration and automatic code generation connect a previously sequential application to gridRuby. 52

5.2 `blast2seq`'s sequential workflow and the modularization of its distinct tasks. 55

5.3 The hypothetical new workflow of `blast2seq`, which uses modularized functionality. 56

5.4 The YAML configuration file used for integration of BLAST with gridRuby. 57

5.5 The process of calling into client code. 59

List of Tables

2.1 Residue insertion, deletion, and substitution in sequence alignments. 5

2.2 A global alignment and a local alignment on the same sequences. 6

4.1 The number of idle messages received by the master in mpiBLAST-PIO. . . 35

5.1 The speedup and efficiency of `blast2seq` when integrated with gridRuby and run over 8 workers. 61

Chapter 1

Introduction

1.1 Motivation

Often biologists find it useful to compare DNA, RNA, or protein sequences to gain valuable information about relationships among them. Sequence comparisons can be used to build relationship trees that assist in evolutionary placement. In addition, newly identified sequences can be compared to known databases to discover structural roles or the origins of the new sequences. All of these comparisons, and the resulting discoveries, can lead to new insights into developmental and structural properties of organisms in our world.

The National Center for Biotechnology Information (NCBI) recognizes the importance of these discoveries. NCBI creates and maintains sequence databases and implements tools for analyzing these data. By accepting sequence submissions from individual laboratories and by exchanging data with other sequence databases (e.g., European Molecular Biology Laboratory, DNA Database of Japan), NCBI ensures a worldwide collection of information.

Through analysis of this information, biologists can gain an understanding of our genetic past and can mold our future. For instance, bioinformatics has lead to a better understanding of issues like hereditary nonpolyposis colorectal cancer (HNPCC), which arises from an altered gene. By using tools from the Human Genome Project [43], researchers were able to locate the problematic gene and produce a blood test to detect it. Since HNPCC can be cured almost completely when diagnosed and treated early, this discovery is extremely valuable.

Dedicated databases exist to track other biomedical issues as well. Influenza virus databases

help researchers identify variants of recorded virus strains. This process assists in the creation of vaccines and helps researchers understand the spread of animal viruses to humans.

Unfortunately, the process of gathering this sort of information is slow, especially with databases growing in size at a tremendous rate. Heuristic sequence searches, such as BLAST, improve upon the speed of traditional dynamic programming algorithms. However, because of the constantly expanding search space, these applications still cannot provide quick turnaround times for large searches.

Parallelized versions of these applications address this issue by dividing the input data across multiple nodes in a cluster. Individual nodes are able to compute on a small part of the entire problem and results can be merged when the search is complete. While this strategy does improve turnaround times, current solutions require tightly-knit environments to run. Further, these solutions are typically hand-crafted around existing sequential applications and cannot be used to parallelize other search algorithms.

This thesis analyzes the performance of an existing parallel sequence search application, named mpiBLAST. This analysis suggests that hand-tuned communication strategies are unnecessary for providing the improvement in speed that mpiBLAST achieves. Thus, we present a framework, named gridRuby, which acts as a generic parallelization system for many types of sequence searches. gridRuby parallelizes a slightly modified sequential BLAST application and achieves super-linear speedup without using advanced techniques like those employed by mpiBLAST.

1.2 Goals

The goals of this work are as follows:

- Provide a high-level view of existing sequence search applications
- Analyze the design and performance of multiple versions of mpiBLAST, an existing parallelized sequence search application
- Motivate the need for a more generic solution to sequence search parallelization and show that such a solution is feasible

Chapter 2 addresses the first goal by reviewing traditional methods of sequence searching as well as modern approaches that improve speed. The design and algorithmic evolution of mpiBLAST is discussed in chapter 3, while its performance is considered quantitatively in chapter 4. The gridRuby framework along with the integration of an application is considered in chapter 5. Chapter 6 concludes the thesis.

Chapter 2

Sequence Searching with a Single Processor

2.1 Basic Sequence Alignment

Biologists use alignments to compare amino acids and protein sequences. Residues of the sequences are represented by letters, and alignments are created by matching these letters as closely as possible between two sequences. Often, it is useful to insert gaps into sequences or match dissimilar residues to create better alignments. Table 2.1 shows the benefit of employing such strategies.

A common way of measuring the similarity of sequences is by using a matrix of residue pair values in which every match has an associated score. Exact matches and likely replacements receive positive scores, while unlikely alternatives receive negative scores. Common protein substitution matrix collections include PAM [13] and BLOSUM [22]. Further, within these sets, certain matrices may be more appropriate than others depending on the lengths of the sequences being compared and the purpose of the comparison [1].

In addition to calculating scores for matches and substitutions, scores may be determined for entire gaps (which may consist of multiple adjacent null characters). Long gaps are penalized only slightly more than short gaps, as a single biological mutation often modifies more than one residue. The most common method of scoring gapped alignments involves the use of affine gap costs, which penalize fairly heavily for the presence of a gap and charge a smaller

Table 2.1: Residue insertion, deletion, and substitution in sequence alignments. Vertical lines between sequences signify matches. Though the two protein sequences being compared are similar, the "unmodified" comparison fails to show this, as it only matches the heads and tails of the sequences. The gapped comparison shows how similar the sequences actually are. Note that the first modification in the gapped comparison is a residue *insertion* while the second modification is a residue *deletion*.

<table>
<tr><th></th><th>"Unmodified" Comparison</th><th>Gapped Comparison</th></tr>
<tr><td>Sequence 1</td><td>TNNSVRKSILLGTFDELRVFI</td><td>TNNSVRKSILLGTF-DELRVFI</td></tr>
<tr><td></td><td>||| | |||||</td><td>||| |||||||||| | |||||</td></tr>
<tr><td>Sequence 2</td><td>TNNVRKSILLGTFPDQLRVFI</td><td>TNN-VRKSILLGTFPDQLRVFI</td></tr>
</table>

amount for each residue the gap contains.

Depending on the reason for comparison, different types of alignments may be desired. If the input sequences are thought to be related over their entire lengths, a global alignment is used, as it requires the complete matching of the pair of sequences. The Needleman-Wunsch algorithm is an example of a global alignment algorithm. However, if the sequences are thought to be only partially related, a local alignment is performed, as it matches only the most similar segments of the two sequences. The Smith-Waterman, FASTA, and BLAST algorithms are examples of local alignment algorithms.

2.2 Smith-Waterman and Needleman-Wunsch

Once the alignments are constructed, the next course of action is to determine which ones scored the highest. Given a scoring matrix, these optimal alignments can be determined using existing dynamic programming algorithms, depending on the desired comparison. The Needleman-Wunsch algorithm [39] computes optimal global alignments, while the Smith-Waterman algorithm [47] computes local alignments. Since similarities between sequences can often be seen only in segments of the input, Smith-Waterman based algorithms are the most popular.

The two algorithms vary only slightly in their implementations. While the Needleman-Wunsch algorithm "remembers" and accumulates the adverse effect of pairing two particular sequences, the Smith-Waterman algorithm "forgets" this effect and is thus more willing to

accept high-scoring, though disjoint, matchings of segments. Table 2.2 shows how this can affect alignments.

Table 2.2: A global alignment and a local alignment on the same sequences. The Needleman-Wunsch algorithm attempts to align the two sequences over their whole lengths, resulting in groups of small numbers of matching residues. The Smith-Waterman algorithm locally aligns the sequences, resulting in larger matched residue groups.

Global Alignment	Local Alignment
`FTFTALILLAVAV`	`FTFTALILL-AVAV`
`F--TAL-LLA-AV`	`--FTAL-LLAAV--`

Current Needleman-Wunsch and Smith-Waterman algorithms run in O(N*M) time, where N and M are the lengths of the two sequences being compared. Both can handle affine gap costs at a small constant-factor decrease in speed [21].

2.3 Heuristic Approaches

Though the Smith-Waterman algorithm provides a relatively efficient method for aligning sequences, if comparisons are done on a regular basis, its dynamic-programming approach is too slow. In time critical situations, the ability to quickly compare against a large database of sequences is required. Hence, a new series of heuristic algorithms emerged.

2.3.1 FASTP and FASTA

The FASTP program [34] employs one such algorithm and improves on the speed of traditional Smith-Waterman based applications. This improvement is achieved by finding identities between two sequences, storing them and their associated offsets in a lookup table (e.g., a hash table), and computing scores based on differences between the offsets. This strategy can greatly reduce the collective problem size, thus reducing the number of comparisons made. The FASTP algorithm is similar to one described previously in the literature [53], but includes multiple modifications and supports the use of a substitution matrix.

The more commonly used FASTA application [40] improves the sensitivity of FASTP with an insignificant degradation in speed, and offers "on-the-fly" translation of DNA sequences for

comparison with proteins. Both algorithms locate identical segments, score these segments and keep those above a threshold, eliminate segments that are unlikely to be part of the final alignment, and use dynamic programming to produce an alignment which incorporates the high scoring segments.

2.3.2 BLAST

The desire to provide quick turnaround times resulted in the development of a new tool, the basic local alignment search tool (BLAST) [2], for the efficient comparison of sequences. Like FASTA, BLAST effectively reduces the problem size by considering similarity at different granularities. However, unlike FASTA, BLAST directly approximates the likelihood of chance matches and evaluates the statistical significance of its results [26], resulting in an order of magnitude speedup in some cases.

To approximate the likelihood of chance matches, the BLAST algorithm is based on finding the maximal segment pair (MSP) in two sequences. A maximal segment pair is defined as the pair of equal-length segments that exhibits the highest score as determined by a score matrix. Such a segment pair produces a score that cannot be improved by reducing or extending the length of the pair. A segment pair is considered a high-scoring segment pair (HSP) if it is locally maximal (that is, it is an MSP) and a Smith-Waterman variant scores it above a specified threshold.

Using Karlin-Altschul statistics [25], BLAST can predict the lowest MSP score at which similarities are still meaningful. Using this knowledge, the algorithm spends less time in regions of sequences that are unlikely to exceed this score. By scanning for small exact matches and extending them and using the calculated MSP score threshold, BLAST is able to effectively approximate the results of dynamic programming algorithms and gains a considerable amount of speedup.

The BLAST algorithm was further improved to support gapped alignments and to increase its speed [3]. Since most computation time in the original BLAST is spent extending exact matches, the new algorithm introduces a “two-hit” method. This strategy requires two exact matches to be within a certain distance of each other before computing an extension. The score threshold must be lowered to produce more hits and achieve the original sensitivity, but since only a small portion of these hits are extended, overall execution time is decreased.

2.4 Other Work

Though Smith-Waterman, FASTA, and BLAST represent a set of popular algorithms used to conduct sequence searches, many more applications exist which serve the same purpose. Some of these applications (e.g., NCBI BLAST, WU-BLAST, and FSA-BLAST) use similar strategies at a basic level, but provide varying functionalities (e.g., FSA-BLAST may be faster than NCBI BLAST, but it does not support translated searches).

The National Center for Biotechnology Information (NCBI) [15] maintains a version of BLAST (including a very large infrastructure) implemented in C and C++. Its design was the basis for the BLAST algorithm described above.

However, there exist quite a few other implementations and variants of BLAST. One well known version, WU-BLAST [19], claims to be the original BLAST application to support gapped alignments with statistics. It is a product of Washington University, carries a copyright, and cannot be modified without written permission. Another, FSA-BLAST [9, 8], guarantees accuracy at twice the speed of NCBI's BLAST. However, it does not support translated searches. BLAT [28], the "BLAST-like alignment tool", is an extremely fast, yet less sensitive, variant of BLAST.

Other applications take different approaches than BLAST to reduce search time. Instead of extending exact matches, PatternHunter [35] uses seeds of non-consecutive letters and extends matches (like BLAST's "two-hit" approach, but more generic). PatternHunterII [32] makes use of multiple seeds to improve sensitivity and speed, and tPatternHunter [30] supports translated searches. Though their ability to match patterns of non-consecutive letters is extremely flexible and often faster, it is difficult to choose appropriate seeds.

To further improve the performance of sequence search algorithms, database pruning algorithms and indexers [24] are used to reduce the search space a-priori. By determining lower bounds for edit distances between strings (essentially, the number of mutations to transform one string into another), these algorithms are able to prune databases significantly. Such approaches typically reduce disk I/O, memory demands, and CPU time, but may discard important potential matches.

There is no optimal solution to the sequence searching problem, as trade-offs in algorithm design must be made. However, these algorithms do share a collective goal: to conduct sequence searches as quickly and accurately as possible.

Chapter 3

Sequence Search Parallelization in Tightly-Coupled Environments

3.1 Introduction

Though local alignment algorithms have come a long way since the original description of Smith-Waterman, performing searches in serial is still too slow. With databases of known sequences growing exponentially, these implementations struggle as the problem size increases over time. Often, the memories of conventional machines fail to store the entirety of large sequence databases, resulting in heavy virtual memory requirements and massive amounts of paging.

Hence, many biologists have turned to parallel computation to resolve the intense CPU and memory demands of their applications. By using multiple machines to accomplish their tasks, their collective memory can be used to avoid paging. Further, much of the computation can be overlapped, resulting in a large improvement in speed.

In this chapter and the next, mpiBLAST, a parallel version of BLAST, is used as a case study to provide insights on the implementation, performance, and problems of parallel sequence-alignment algorithms. This chapter looks at mpiBLAST from an algorithmic and implementation perspective, while the following chapter evaluates its performance through experimentation.

3.2 Conventions

To keep discussions of these applications and their techniques consistent, the following definitions are used:

- A *database* is a collection of sequences which may be in one of many different formats. NCBI and other organizations have collections of very large databases, which are often used as input to sequence comparison applications.
- A *query* is a single sequence that is compared to an entire database.
- A *query set* is a collection of sequences that is compared to an entire database. Often, biologists find it useful to compare a collection of queries against a database rather than using only one query per run, hence they create a query set.
- *Database segmentation* involves dividing a database into smaller components, so that each node of a cluster or processor of an SMP machine can search on these components in isolation (Figure 3.1). This process takes advantage of parallelism as well as the collective memory of a cluster, such that the pieces of a large database are stored in the memory of each node.
- *Query segmentation* involves dividing a query set into smaller parts (Figure 3.1) for similar reasons. Nodes of a cluster or processors of an SMP system can search parts of the query set against an entire database in parallel. Note, however, that if the database replicated to each node is larger than the memory of that node, paging issues will create slowdown in execution.
- *Fragments* are the fractions of a database or query set which result from segmentation (e.g., database segmentation produces database fragments). In this thesis, *chunk* is synonymous with fragment.

3.3 mpiBLAST

mpiBLAST is an open-source, parallelized version of NCBI's BLAST, which makes use of MPI [36] for communication. Though the original realization of the algorithm simply made

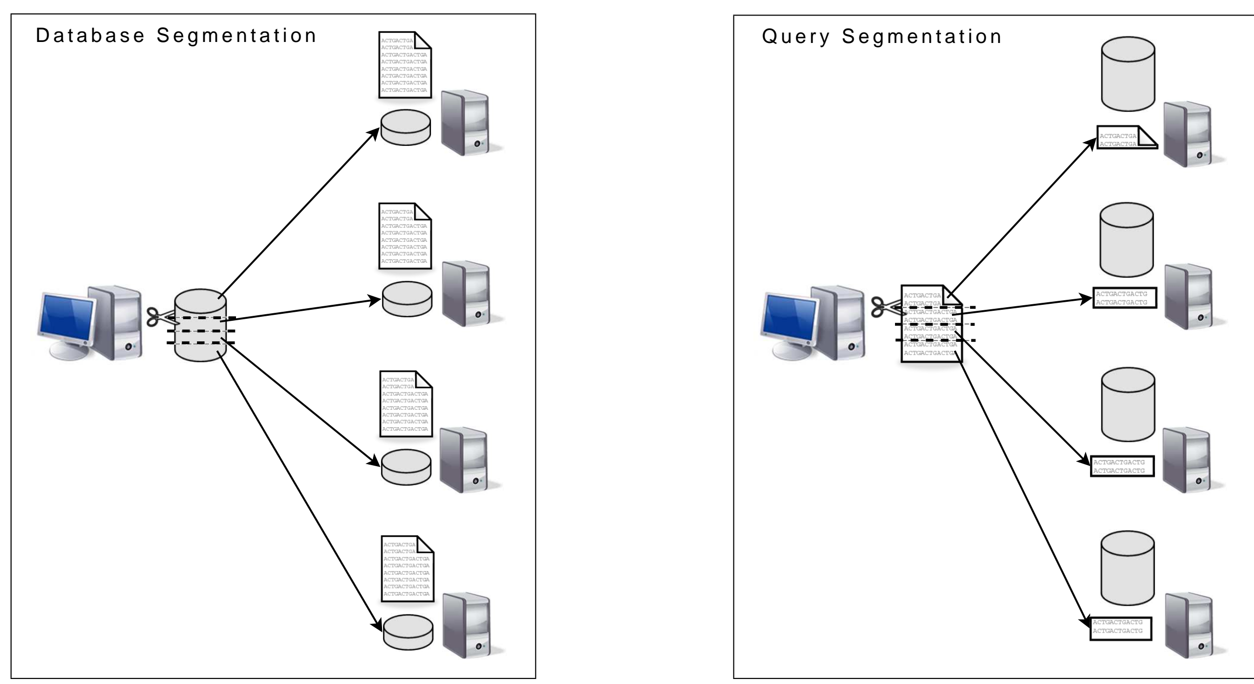

Figure 3.1: Typical database and query segmentation patterns. In database segmentation, each node searches the entire query set against a fragment of the database. In query segmentation, each node searches the entire database using only a portion of the query set.

use of database segmentation to ensure that the search at each node is executed "in memory," mpiBLAST achieves super-linear speedup. Further, over the course of a few years, it has seen many modifications which help improve its performance, reliability, and scalability. The following sections trace mpiBLAST's evolution up to the current version.

3.3.1 mpiBLAST-1.2

NCBI BLAST provides two primary applications used for searching a sequence database with a set of queries. The first application, `formatdb`, formats a given database to help speed up searches. Further, if the entire database will not fit into the memory of the machine, `formatdb` can split the database into chunks; in this way, the application can run on smaller chunks (which fit into memory) one at a time. The second application, `blastall`, performs the actual searches.

The first introduction of mpiBLAST (version 0.9) [12] included additional applications. `mpiformatdb` wraps the NCBI `formatdb` and provides additional functionality by creating small fragments from the specified database for use with mpiBLAST and moving these fragments to storage shared among all nodes that will use the fragments. The other application, `mpiblast`, is a parallelized version of BLAST that uses database segmentation to achieve speedup; the actual BLAST searches are handled by NCBI toolkit code, while parallelization is achieved through supporting code and MPI.

mpiBLAST makes use of a master process, which distributes database fragments to worker processes. The master process handles tracking of fragments, sending fragments to workers, and collecting results. Workers simply run the actual search.

Worker processes start by informing the master of which database fragments they are already storing locally. The master then reads the query set from disk and broadcasts it to all of the workers. Once workers have received the query set, they send an idle message to the master. When the master receives an idle message, it assigns the worker a database fragment to copy or search. After the worker finishes with this task, it sends the master an idle message. As long as there are database fragments that have not been searched, this process is repeated.

With mpiBLAST-0.9, the master collects and sorts partial results from the workers. Then, to gather the necessary information (like sequence data) for output, the master reads from the database on shared storage. Once the data are converted into an appropriate format,

they are output and execution is finished.

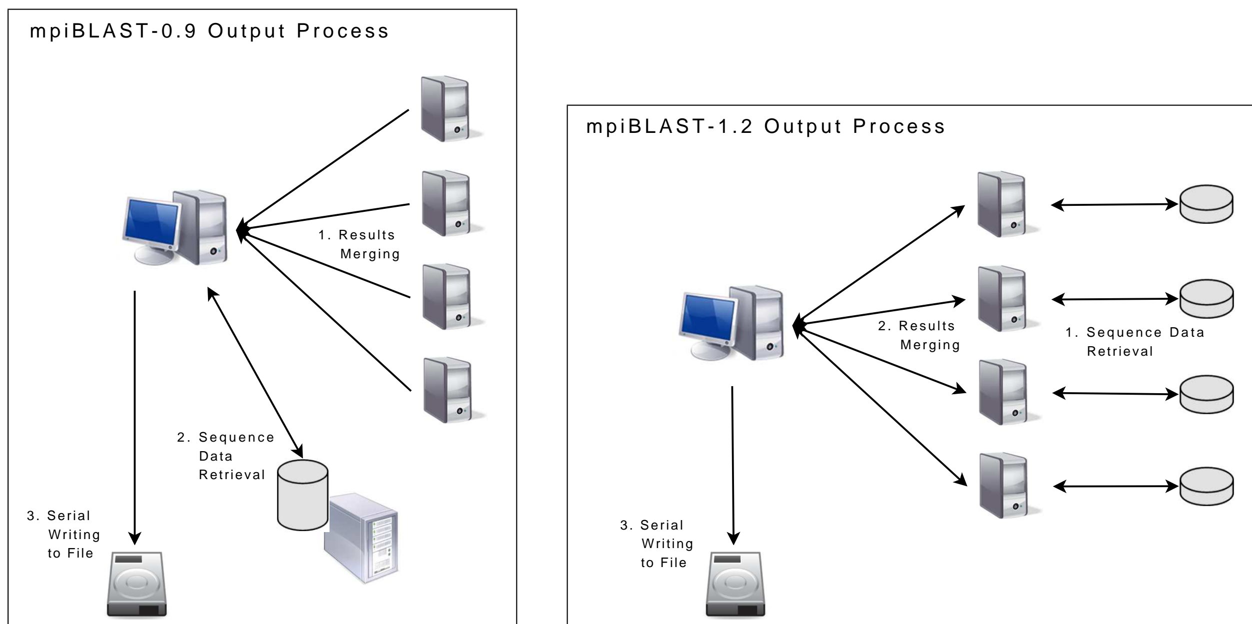

Figure 3.2: mpiBLAST-0.9 and mpiBLAST-1.2 output processes. In mpiBLAST-0.9, the master collects all partial results from workers, looks up sequence information from the database on shared storage, formats the results, and writes the results to file. In mpiBLAST-1.2, the master queries workers for results and the workers look up the associated sequence data from their local database fragments. Then, results are sent to the master for merging, formatting, and writing. Allowing workers to look up sequence information makes use of data locality (as the workers are storing database fragments locally) and increases parallelism (as workers can retrieve information from their respective fragments in parallel).

mpiBLAST-1.2 operates similarly to mpiBLAST-0.9, though its output process is slightly improved (Figure 3.2). After low-scoring results are filtered out, sequence information is obtained by the master from workers rather than from shared storage. This allows the master to query workers for data from their local database fragments, convert the results into the necessary format, and write the data to disk in serial.

3.3.2 pioBLAST

While the first versions of mpiBLAST improved upon BLAST's performance considerably, pioBLAST [33] addresses some issues with mpiBLAST to improve performance and scalability. pioBLAST makes use of the following techniques:

- On-the-fly database fragmenting
- Caching of important data
- Parallel I/O on shared files
- Refined processing of results

The static database partitioning scheme used by mpiBLAST is restrictive. Using fewer database fragments than the number of nodes for a search poorly uses resources, as compute power goes to waste. Further, using more fragments than the number of resources increases both search and non-search time, due mainly to collecting, merging, and outputting results. Users with multiple sets of resources (i.e., clusters with differing numbers of nodes) are required to collect multiple formatted databases when using static partitioning. In these scenarios, users are burdened with database maintenance, and valuable disk space goes to waste in storing these, often large, databases.

pioBLAST resolves these issues by providing a dynamic database-partitioning scheme. MPI-IO [51] is used to access shared databases in parallel. The master process distributes file offsets, which represent virtual database fragments, to worker processes at execution time. The workers can then read the fragments into memory, in parallel, and perform their search as they normally would.

In addition to the general restrictiveness of static database partitioning, mpiBLAST-1.2 suffers from the effects of serializing results data. In mpiBLAST-1.2, workers send partial results to the master for sorting. After sorting the results by score and determining which sequences will appear in the output, the master requests other information (such as the actual biological sequence data) related to the partial results sent by each worker. The workers consult their local database fragments on disk for this information and forward it on to the master.

pioBLAST uses a strategy different than mpiBLAST-1.2 to improve the efficiency of this process (Figure 3.3). When results are discovered, workers cache potentially useful sequence information so it need not be retrieved from disk again later. Then, as is the case with mpiBLAST, pioBLAST sends partial results to the master for sorting. However, once the sorting is complete, the master does not request further information from the workers for output. Instead, the master notifies the workers of which results should be included in the

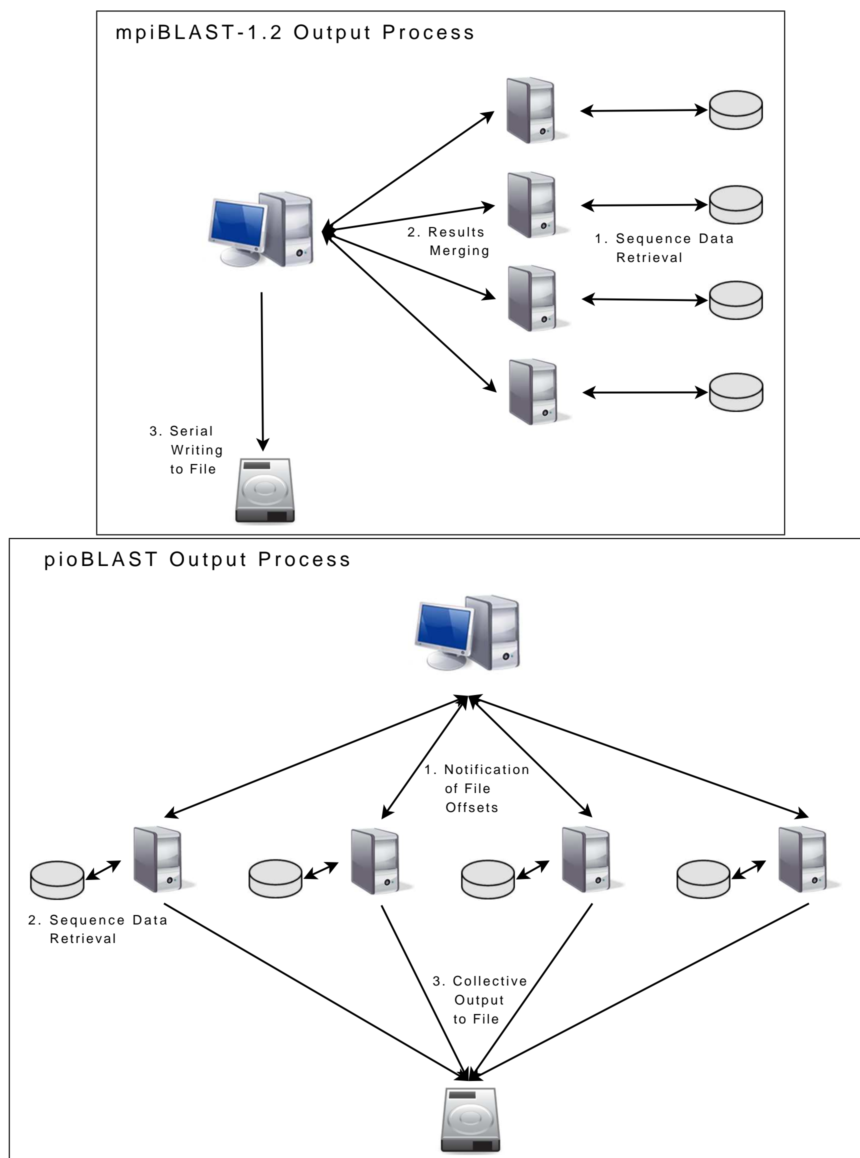

Figure 3.3: mpiBLAST-1.2 and pioBLAST output processes. In mpiBLAST-1.2, the master queries workers for results and the workers look up the associated sequence data from their local database fragments. Then, results are sent to the master for merging and writing. In pioBLAST, the master informs workers of which results to output and where in the file to output them. Then, workers retrieve sequence data from their local database fragments, format the results, and output to the file collectively.

output, and the workers do the actual writing. This process eliminates the extra transfer of sequence data to the master and allows workers to use their cached data to reduce reads from disk.

pioBLAST makes its final improvement in the output step. The mpiBLAST-1.2 master serializes the process of collecting sequence data and outputting final results. On the other hand, given a parallel filesystem, pioBLAST uses MPI-IO to effectively collect and write results from the noncontiguous memory of the workers. This takes advantage of the previous performance enhancement (locality of sequence data on the worker) as well as conducting output in parallel.

3.3.3 mpiBLAST-1.4

In parallel to the pioBLAST work, a new version of mpiBLAST emerged. mpiBLAST-1.4 was produced to resolve some of the issues discovered with mpiBLAST-1.2. This work includes the following changes:

- More confidence in the accuracy of alignment score calculations
- Reduced memory demands on the master
- Reduced load on network storage
- Use of query segmentation as well as database segmentation

In addition to aligning two sequences, mpiBLAST produces a score for each alignment, which indicates how similar two sequences actually are. These scores are used to sort results, so the most relevant matches are found at the beginning of mpiBLAST's output. Prior to mpiBLAST-1.4, these scores were approximated and did not exactly match the scores produced by NCBI's sequential BLAST.

To resolve this issue, the mpiBLAST-1.4 master collects information about the database and queries, and broadcasts these data to the workers before any searching takes place. Since the database is fragmented between workers, this initial database scan is necessary to provide workers with an absolute view of the problem size so that they can accurately calculate the relative significance of matches.

mpiBLAST-1.4 also corrects an issue that was noticed when copying database fragments from shared storage over NFS. When a large number of workers started up, all of them would attempt to copy a database fragment to their local disk. However, this creates a substantial amount of load on the NFS server and results in slowdown (or failure) of the copies. Instead, mpiBLAST-1.4 allows only a certain number of workers to copy database fragments at the same time, thus reducing load on the storage server.

In addition to addressing issues with mpiBLAST-1.2, mpiBLAST-1.4 introduces query segmentation. This process involves dividing the query set among workers, so each worker can search against an entire database in parallel. With sufficiently small databases (such that the whole database fits into a node's memory), query segmentation can produce results similar to those achieved by database segmentation.

Further, this technique can help distribute workload when used in conjunction with database segmentation. Previous work [12, 33] notes that heavily fragmenting a database introduces overhead and extends execution time, which makes a strong case for using other techniques, such as query segmentation, for data decomposition. By duplicating certain fragments across nodes, and dividing queries between them, the overhead associated with database fragmentation can be reduced.

Query segmentation can also assist in balancing load if certain database fragments require long times to search. If workers finish with their database fragment, they can take on the fragment of another worker and search different queries against it. In this way, if there is a large discrepancy in the finish time of workers, they can assist each other in completing the job.

Pedretti et al. note the usefulness of multiple levels of parallelism (including the combination of database and query segmentation) in previous work [41].

3.3.4 mpiBLAST-PIO

The latest member of the mpiBLAST lineage is mpiBLAST-PIO. The code-base derives from mpiBLAST-1.4, so it incorporates the solutions to mpiBLAST-1.2's problems, as well as query segmentation. Further, it makes use of some strategies from pioBLAST, such as using workers to process output and to write in parallel (Figure 3.4) if the underlying filesystem supports it.

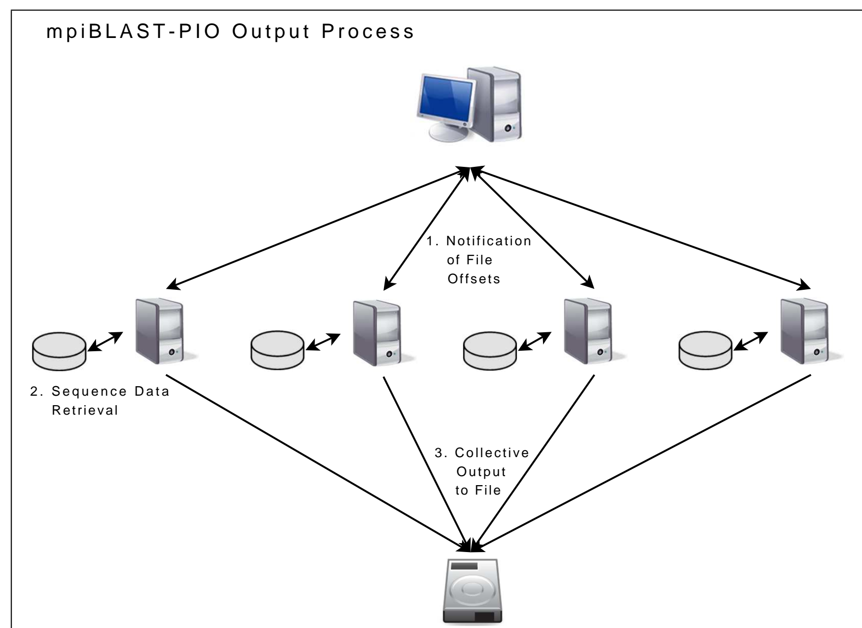

Figure 3.4: mpiBLAST-PIO output process. As with pioBLAST, workers output to the results file collectively. Note that even if the filesystem lacks support for parallel operations, this process is still more efficient than that of mpiBLAST-1.2, as workers in mpiBLAST-PIO conduct more work in parallel (i.e. workers format their own results rather than relying on the master to do it).

3.4 Other Work

The NCBI BLAST toolkit itself provides support for multi-processor computation on SMP systems using database segmentation. Processors within such a system search on separate portions of the database, and results are collated as one. This strategy, when coupled with an application like mpiBLAST, can be used to further increase parallelism.

Proprietary cluster-based implementations are available as well, including TurboBLAST [6] (which is now inactive), Hyper-BLAST [29], PowerBLAST [42], and Paracel BLAST [7]. The published performance evaluation of Hyper-BLAST was conducted on an 8-node cluster, while PowerBLAST and Paracel BLAST fail to provide accessible numbers. Also, the commercial PowerBLAST should not be confused with the PowerBLAST [55] from NCBI, which uses various filtering techniques (but not parallelization) to improve the speed of BLAST.

In addition to software-level optimizations, effort has been spent in improving BLAST-based applications by computing sequence alignments in hardware. The first implementation of this strategy is seen in BioSCAN [46, 52], a VLSI-based approach to sequence alignment. More recently, multiple FPGA-based solutions have emerged, including RC-BLAST [38] and a single-pass version of BLAST by Herbordt et al. [23]. These solutions could be replicated and used in conjunction with software-level parallelization techniques to take advantage of multiple levels of parallelism.

Chapter 4

Analysis of Parallel Sequence Search in Tightly-Coupled Environments

4.1 Introduction

To gain a quantitative understanding of the performance of mpiBLAST and the practical differences between versions, merely reviewing its algorithms is not sufficient. Unexpected factors like faulty network filesystems or unforeseen communication patterns can disrupt the operation of even a well-crafted application. Testing applications in a real environment can uncover subtle issues and lead to solutions to problems, resulting in a more robust system.

Further, no formal analysis has been conducted since mpiBLAST's introduction, thus all efforts to improve performance have been based on speculation. Discoveries from a careful analysis can lead to insights regarding past decisions, as well as possible future directions.

Therefore, we have profiled two versions of mpiBLAST (1.2 and PIO) in an attempt to identify their strengths and weaknesses and to quantify key differences.

4.2 Conventions

The following definitions are used in the remaining sections:

- *Speedup* represents how much faster mpiBLAST is compared to its sequential counterpart. All speedup numbers are calculated by dividing the execution time of the sequential application (i.e., `blastall`) by the execution time of mpiBLAST.
- *Efficiency* represents how much speedup is achieved by each worker in mpiBLAST. All efficiency numbers are calculated by dividing the execution time of mpiBLAST by the number of workers used.
- A *phase* represents a series of steps in the execution of mpiBLAST. The time spent in a phase is accumulated over the entire execution of the program, so mpiBLAST may enter a phase multiple times. An example phase of mpiBLAST is the sending of results from workers to the master.

4.3 Experimentation Variables

By fixing a number of experimentation variables, we can identify the impact of differences between mpiBLAST versions. The hardware platform and environment, input data, and code modifications can greatly affect the runtime of an application, and therefore must be chosen carefully. These issues are discussed in the following sections.

4.3.1 Hardware

All of the following experiments were conducted on System X, the 1100-node supercomputer located at Virginia Tech in Blacksburg, Virginia. Each compute node is an Apple Xserve G5, housing dual 2.3-GHz PowerPC 970FS processors, four gigabytes of DDR400 RAM, an 80-gigabyte SATA hard drive, and an InfiniBand host channel adapter. Further storage is provided by an Apple Xserve RAID over NFS, yielding almost three terabytes of storage.

System X was chosen as the platform for experimentation due to the large number of nodes, which provided the necessary foundation for analyzing scalability. Nodes are allocated to users through a job submission queue. However, note that this does not provide any guarantees regarding network or NFS storage performance, as other users on the system could be using these resources. Thus, multiple runs of each trial were conducted to improve confidence in the accuracy of the results.

4.3.2 Data

Both NCBI BLAST and mpiBLAST take a query set and a database file as input. Choosing these parameters affects the total execution time as well as where time is spent in both applications. Different input sets may reflect inconsistently on either application's performance, and they must be carefully considered to match typical patterns of use.

To accurately represent the scenario in which a large database is used as context for newly annotated sequences, the GenBank `nt` database was chosen. By comparing a gene to a set of known sequences, relevant information regarding the gene's purpose can be obtained. A large database serves this purpose well, as valuable information can be gleaned from the potentially large number of resulting matches.

The chosen query set is the same as the one used in [12].

4.3.3 Toolkit

Both NCBI BLAST and mpiBLAST make use of the NCBI BLAST Toolkit, which is written in the C language. Considering requirements such as availability and interface, the toolkit released in October 2004 fit best and was used by both applications in the presented experiments. Further, the sequential application released with the toolkit, `blastall`, implements an algorithm that allows for the efficient searching of large sizes of input. Later versions of the application employ a new algorithm, resulting in quicker feedback but at the cost of problem size scalability. Hence, the toolkit from 2004 provides the ability to compute on the large `nt` database sequentially, which is essential for calculating speedup.

The sizes of modern databases and memories restrict the possibility of keeping searches strictly in memory. With sizes of databases doubling every 12 months [5] and memory capacities quadrupling only every 36 months (i.e. doubling every 18 months), this scenario is likely to continue. Further, due to swap space limitations, the `nt` database does not even fit into the cumulative physical and virtual memory of a System X node.

Hence, the `formatdb` application is used to divide large databases into smaller chunks, so that sequential applications (which run on a single node) can search them. In this way, applications like `blastall` are able to execute on the small fragments and produce results progressively.

4.3.4 Method of Profiling

Particularly with high-performance applications, determining the lengths of execution phases is innately problematic as measurement of these events can perturb their length. Care has to be taken so as not to hinder the application by introducing additional functionality. Hence, measures were taken to make profiling of the mpiBLAST code as efficient as possible.

A low-complexity class was introduced to both mpiBLAST versions. This class, once instantiated, provides a method which associates messages with a measured timebase. Timebases, which are represented as `unsigned long long`s, are read from the machine's hardware using the C++ inline assembler; the intent is to measure at a fine granularity while affecting the application's performance as little as possible.

The first logging of a message represents the beginning of a section to be timed, while a second logging of the same message marks the end. Duplications mark the beginning and end of re-executed sections of code (such as the body of a while loop), so timebases for these sections are accumulated. A C++ map is kept to associate accumulated timebases with their messages.

Profile points were added around pertinent sections of mpiBLAST code and care was taken to avoid "over-profiling," as doing so might adversely affect the measurement of events. After execution, a log file is written containing the messages and accumulated timebases. A set of Perl scripts was used to analyze the profiles after execution by converting the timebases to approximate lengths of actual time.

4.4 mpiBLAST-1.2 Analysis

mpiBLAST-1.2 marks a milestone in mpiBLAST's development, as it includes a refined codebase and optimized sequence lookups. Further, it is the last major revision to implement only database segmentation (query segmentation was first introduced in mpiBLAST-1.4). mpiBLAST dramatically reduces the execution time of NCBI BLAST, but an analysis of where time is spent is crucial in understanding how mpiBLAST performs and how it can be improved. Optimally, insights gained from such an analysis can be applied to other parallel search algorithms, thus improving searches as a whole.

All numbers presented for workers in the following sections are computed by averaging the

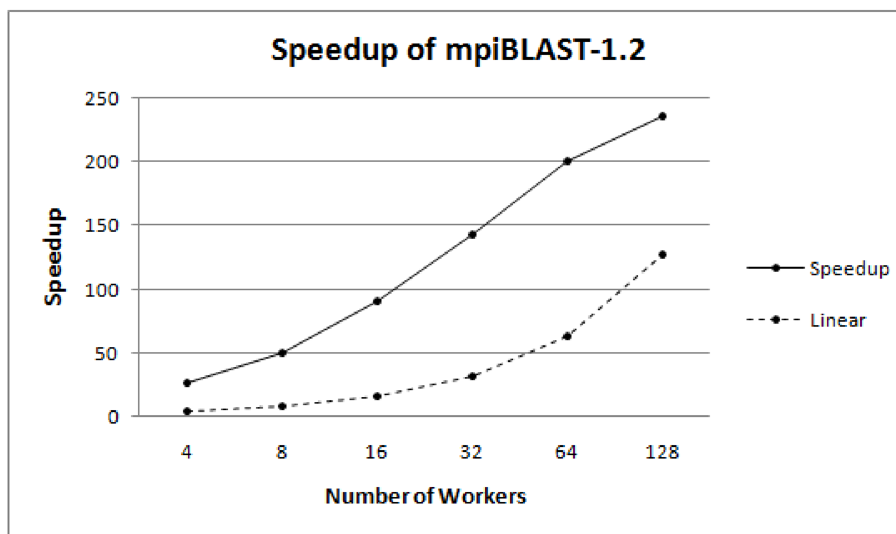

Figure 4.1: Speedup of mpiBLAST-1.2.

values for all the workers within the presented set. For example, when a runtime is reported for "128 workers," the value is an average of all 128 workers' times.

4.4.1 High-level Discussion

The motivation behind any parallel application is the desire to improve performance. Further, the simplest and most obvious metric for evaluating a parallel application is speedup; hence, it is considered first.

The `nt` database was fragmented into parts equal to the number of workers used, and a single fragment was distributed to each node a-priori. Then, workers searched their fragments using a set of queries. This database pre-distribution strategy matches the case in which biologists intend to search a single database multiple times, in an attempt to match many queries against the set of all known sequences. As can be seen in Figure 4.1, mpiBLAST-1.2 gains formidable speedup over NCBI's sequential BLAST.

Figure 4.2 indicates that the execution time associated with the BLAST search alone (excluding everything else) decreases as the number of nodes increases. Therefore, parallelization is successful in reducing the time associated with searching, which results in the speedup just

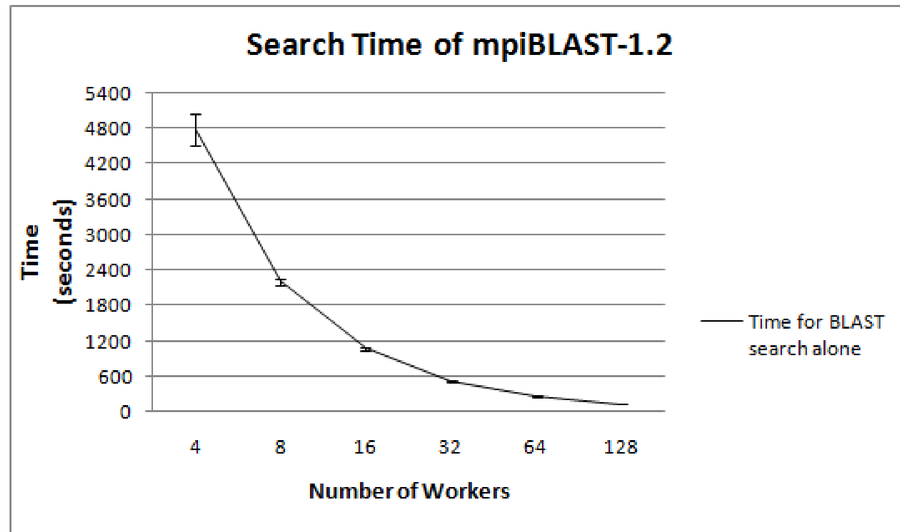

Figure 4.2: Search time of mpiBLAST-1.2. Error bars indicate the standard deviation of worker search times.

mentioned.

At first, this point seems obvious; dividing work onto progressively more processors, without measuring the associated parallelization overhead, should yield shorter and shorter search times. However, since the mpiBLAST code interfaces with NCBI's API, more may happen behind the scenes than is expected. One has to be wary of startup costs associated with invoking a search, especially when running with large numbers of database or query fragments. Though this issue does not arise in the tested cases, it should be mentioned, as future experimentation with larger numbers of nodes could encounter such drawbacks.

Though searching on larger numbers of fragments does not produce a noticeable amount of overhead, the steps necessary for parallelizing BLAST do, in fact, add to mpiBLAST's total execution time. Network-based notifications, scheduling, and data swapping are all necessary to support parallelization. To keep discussions of these issues consistent, the time required by the BLAST search alone will be called *search time*, while the time spent on other tasks (writing, scheduling, waiting, etc.) will be called *non-search time*. Further, those parts of non-search time that are necessary solely for the purpose of parallelization will be called *parallelization overhead*.

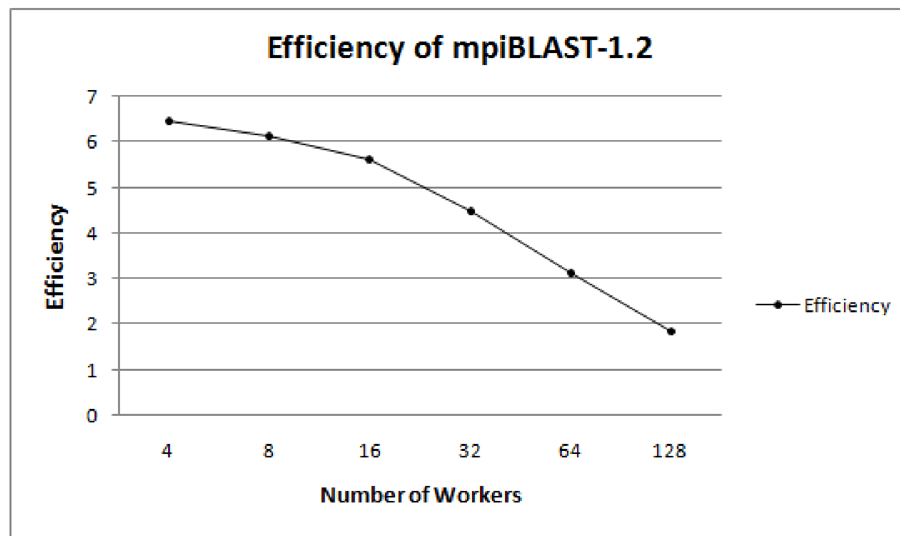

Figure 4.3: Efficiency of mpiBLAST-1.2.

mpiBLAST-1.2 certainly benefits from a large number of workers, but the efficiency of these workers should be considered as well. Figure 4.3 indicates that efficiency drops steadily with the number of workers.

Usually, an efficiency of 1.0 is the optimal case, representing a single-fold speedup per process. That is, optimally, with 4 workers one would expect to solve a problem in a quarter of the time it takes with 1 worker. However, as mentioned previously, when BLAST is run sequentially, a large amount of paging occurs because the entire database does not fit into the memory of a single machine. With 4 workers, each database fragment fits into the memory of each worker, thus eliminating paging issues. Therefore, when run with multiple workers, mpiBLAST sees more than a single-fold speedup per worker.

Though efficiency remains above 1.0 for all cases, the drop between 4 and 128 workers is fairly drastic. Figure 4.4 shows that the efficiency of BLAST search time alone increases with the number of workers. This indicates that tasks other than searching result in the degradation of efficiency per worker.

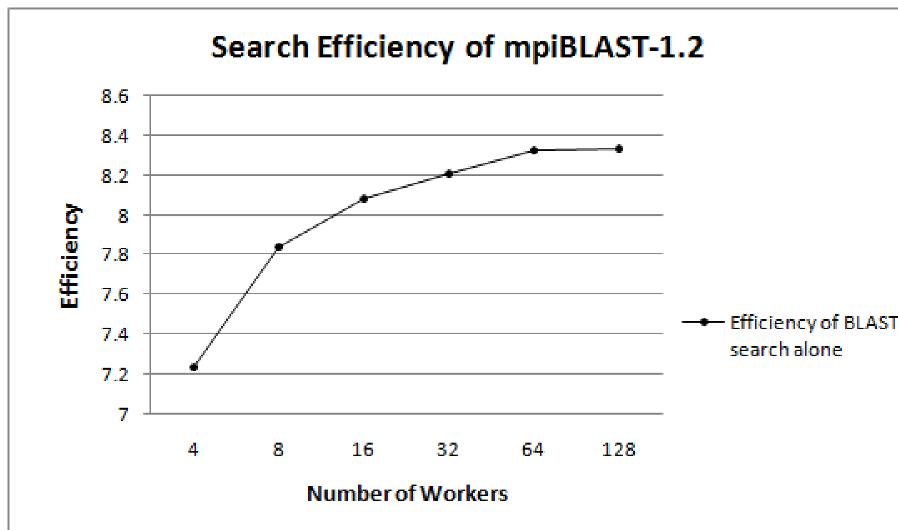

Figure 4.4: Efficiency of the BLAST search in mpiBLAST-1.2.

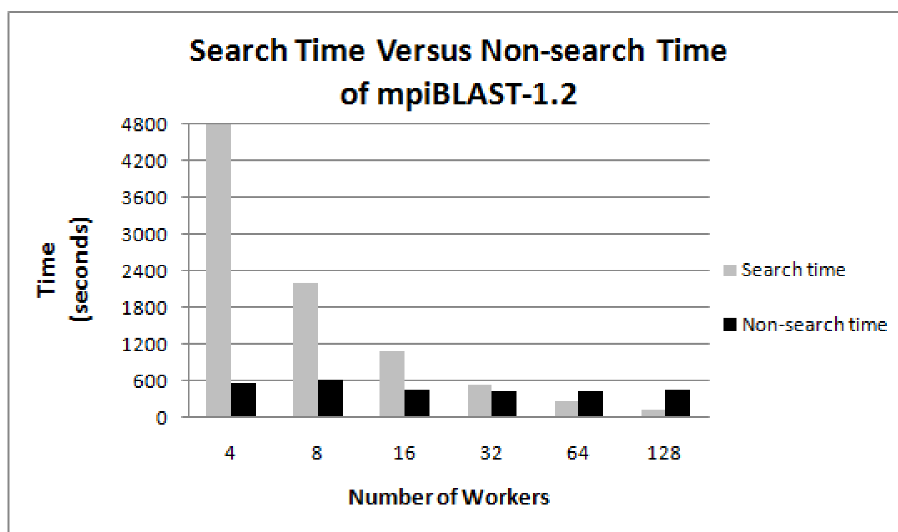

Figure 4.5: Search related and non-search related time of mpiBLAST-1.2.

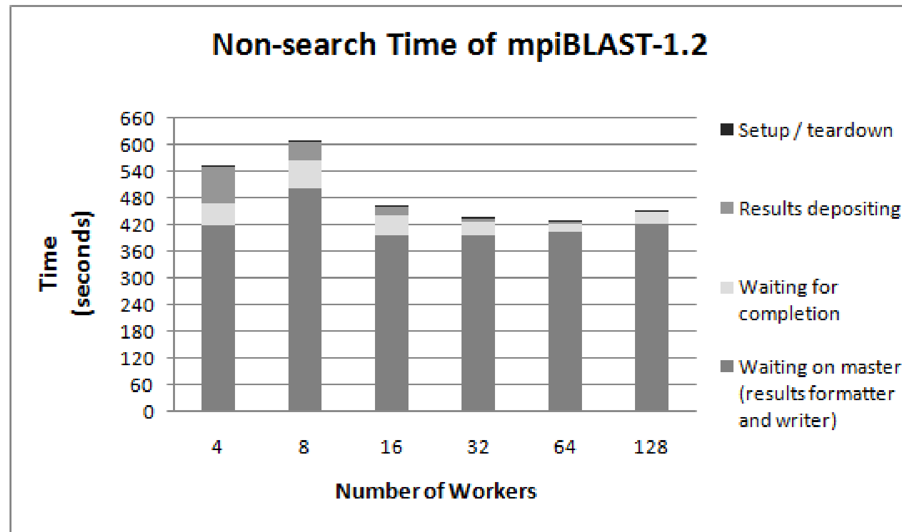

Figure 4.6: Breakdown of non-search related time in mpiBLAST-1.2.

4.4.2 Discussion of Non-search Time and Overhead

It is then important to further analyze the effects of the stages in mpiBLAST's execution. Figure 4.5 compares search time with all other (non-search) time. While drastic increases in non-search time could indicate high scheduling overhead or result collection inefficiencies, this is not the case, as non-search time remains relatively constant for up to 128 workers. The non-search time exceeds the search time above 32 workers; this is due to a decrease in search time and not an increase in non-search time.

The time workers spend conducting non-search related tasks remains fairly constant across all runs, as is shown in Figure 4.6. This figure gives a closer look at the small, "non-search time" bars of Figure 4.5; it includes necessary startup and finish costs (setup and teardown), as well those categories that are considered parallelization overhead (results depositing, waiting, etc.).

The tasks attributed toward "setup / teardown" are composed of general "housekeeping" responsibilities, such as checking arguments, preparing data structures, and performing initializations. As expected, the tasks add very little to the execution time.

After searching their fragments, workers must send their partial results to the master. The

master sorts this information and requests more data from the workers for the results that will be output. The category "results depositing" represents the time it takes for workers to lookup the appropriate sequences from their respective database fragments and synchronously send them to the master. The length of this phase decreases with the growing number of workers since database fragment sizes decrease with more workers (recall that fragment size is equal to the total database size divided by the number of workers). That is, more workers scanning smaller fragments in parallel results in improved speed.

Once workers finish searching a database fragment, they wait to be notified of more work from the master. Since some workers finish their work before others, they simply block until all other workers are finished (as there are no more database fragments to search in the meantime). Further, some workers finish sending their results to the master before others, as the size of the results produced by each worker varies. The time spent idling (while waiting on other workers) during these tasks is represented by the "waiting on completion" category.

This phase contributes a small amount of time to mpiBLAST-1.2's execution. Though search times vary less significantly as the number of workers grows (see Figure 4.2), this phase sees abnormalities in length (at 8 workers and 128 workers) due to differences in sequence lookup times between workers. Only when all sequence data has been sent to the master for results processing and the master has finished outputting, can the workers quit.

Therefore, writing of results by the master creates additional waiting time at the workers. Since workers simply wait for the master to query them for results and sequence information, they wait for a significant amount of time while the master requests, sorts, processes, and outputs results. *This phase, "waiting on master," is the main contributor to time spent in non-search related activities.*

4.5 mpiBLAST-PIO Analysis

mpiBLAST-PIO marks another milestone in the evolution of mpiBLAST. It incorporates query segmentation for the efficient use of computational resources, pipelined copy operations for network storage load balancing, optimized communication to reduce overhead, and parallel IO operations to eliminate serialization of output at the master.

Note again that numbers presented for a set of workers represent the average of the values

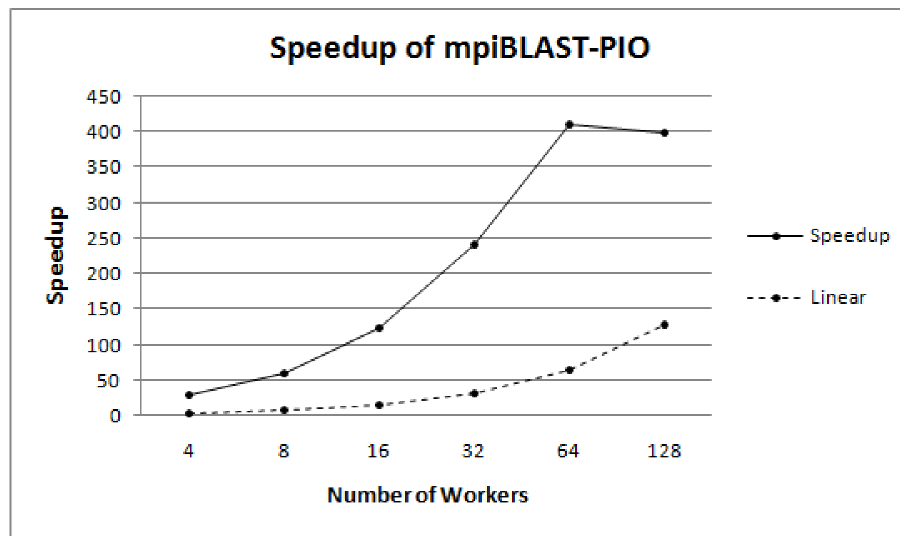

Figure 4.7: Speedup of mpiBLAST-PIO.

for that worker set.

4.5.1 High-level Discussion

In the presence of a filesystem that supports parallel operations, the workers of mpiBLAST-PIO write their search results to a common file on shared storage. Lacking such a filesystem, as is the case with System X's NFS, the workers simply send their results to the master, which sorts the results and writes them to file.

As mentioned previously, this process is an optimized version of the mpiBLAST-1.2 results writing algorithm. Instead of sending the results to the master in their native format (as is done in mpiBLAST-1.2), workers convert their results into a format appropriate for output before sending them to the master. This saves time by conducting more work in parallel and by reducing communication overhead. Rather than querying workers for sequence data necessary for converting raw results, the master simply sorts already formatted results and writes them.

Again, as a baseline for performance analysis, speedup is considered. As with the mpiBLAST-1.2 runs, the `nt` database fragments were pre-distributed to worker nodes before searching.

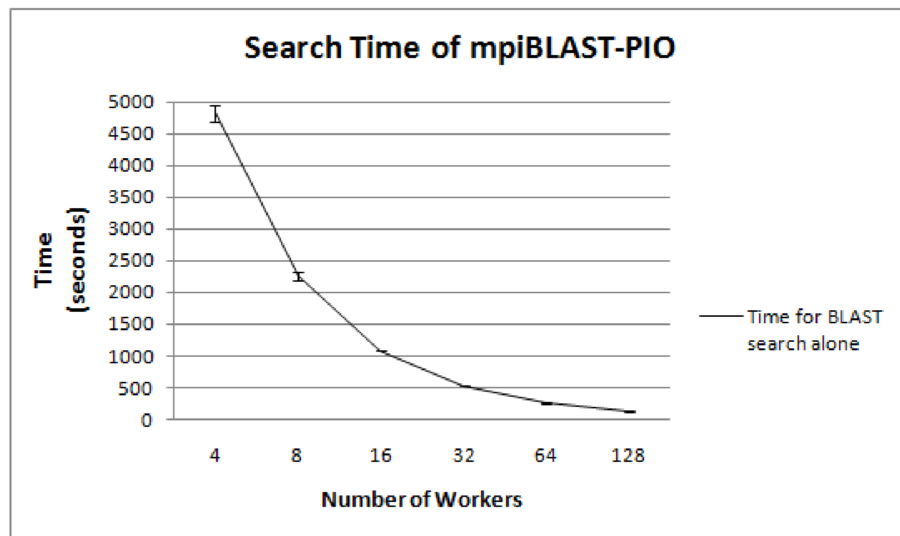

Figure 4.8: Search time of mpiBLAST-PIO. Error bars indicate the standard deviation of worker search times.

Along with matching a real-life scenario, this process avoids measuring the time necessary to simply copy data across the network; the goal of this analysis is to measure the time required to conduct a search in parallel, and not to record the time needed to copy fragments across the network. Given these criteria, mpiBLAST-PIO shows super-linear speedup over sequential BLAST, as can be seen in Figure 4.7.

Figure 4.8 shows that mpiBLAST-PIO reduces the amount of time spent conducting the BLAST search. Again, BLAST search startup costs are small enough so as not to perturb the effectiveness of the parallelization.

Speedup is obtained with all numbers of workers, but a degradation is seen from 64 to 128. Though a 400-fold speedup is seen at 128 workers, execution time increases (speedup decreases) as compared to 64 processes. This leads to some concern regarding the efficiency of mpiBLAST. As can be seen in Figure 4.9, mpiBLAST-PIO efficiently makes use of up to 16 workers, but drops off after this point.

As is the case with mpiBLAST-1.2, non-search time results in a continuous loss in efficiency. And, though the BLAST search alone scales well (Figure 4.10), mpiBLAST-PIO's performance takes a hit with 128 workers. The extra time spent in handling non-search related

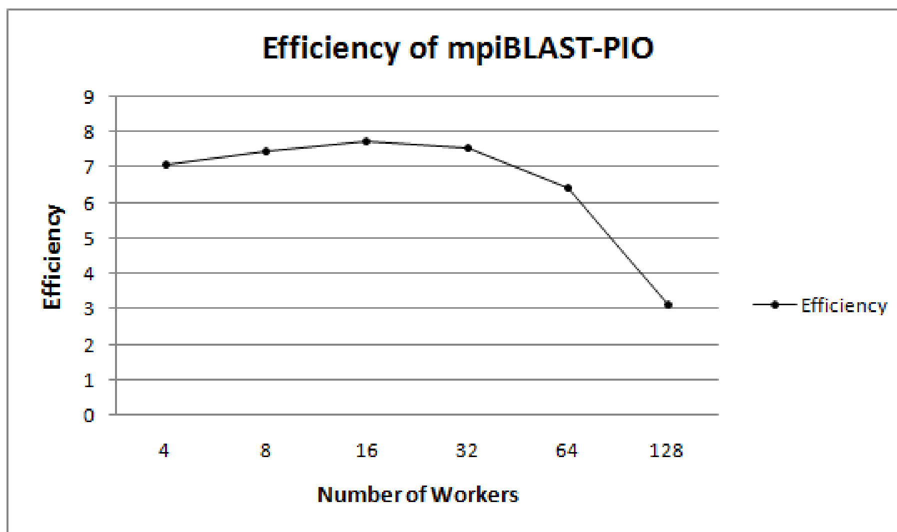

Figure 4.9: Efficiency of mpiBLAST-PIO.

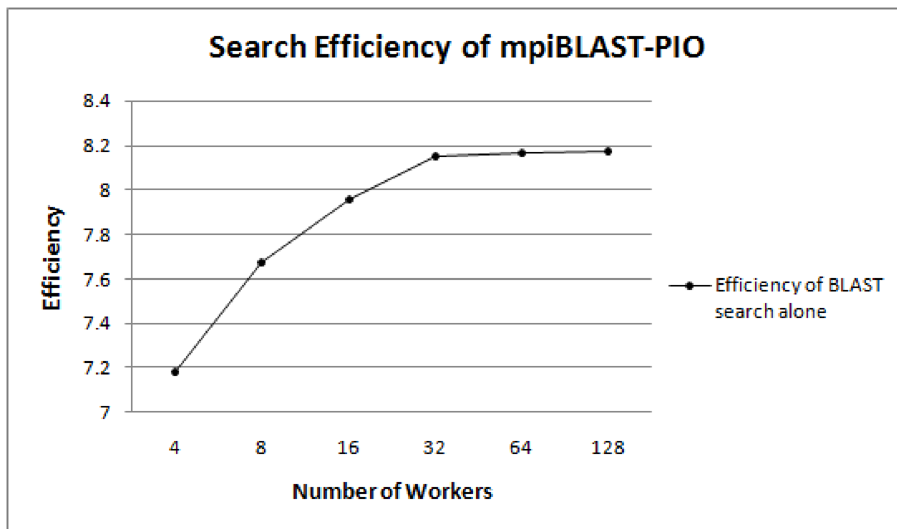

Figure 4.10: Efficiency of the BLAST search in mpiBLAST-PIO.

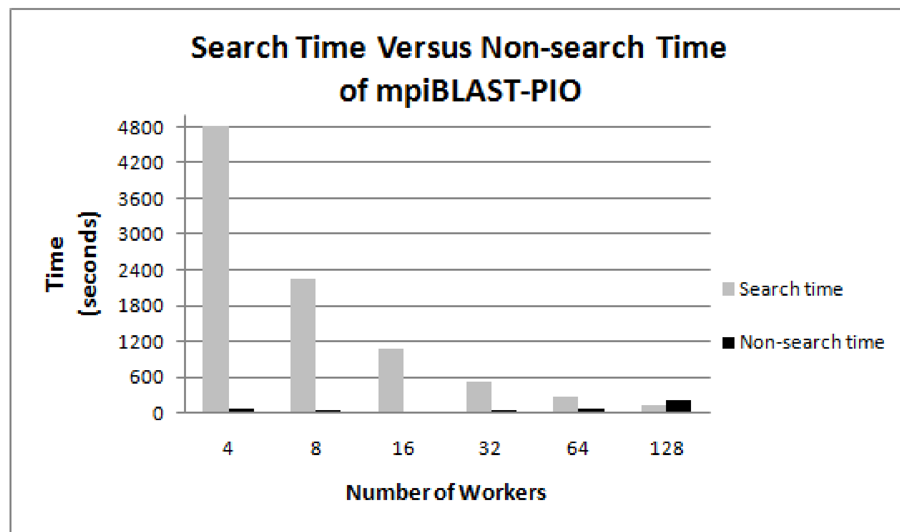

Figure 4.11: Search related and non-search related time of mpiBLAST-PIO.

tasks pushes its cost above that required for searching, resulting in a overall time greater than with 64 workers.

Figure 4.11 compares the time required to run a BLAST search in parallel with the amount of time required for non-search tasks. For small numbers of workers, non-search time is negligible. And even though it constitutes more than half of the execution time in the 128 worker case, non-search time does not increase. Rather, since the search time is reduced significantly by larger numbers of workers, non-search time represents a larger portion of the overall execution time.

4.5.2 Discussion of Non-search Time and Overhead

Though non-search time is relatively small across all runs, it does increase for larger numbers of workers. Figure 4.12 shows the breakdown of non-search time with respect to the number of workers used. It includes tasks that are necessary in the sequential version (setup and teardown) as well as parallelization overhead (waiting, collecting of statistics, etc.).

As with mpiBLAST-1.2, the category labeled "setup / teardown" represents general startup and finish costs, as well as various other supportive tasks. Again, it contributes very little

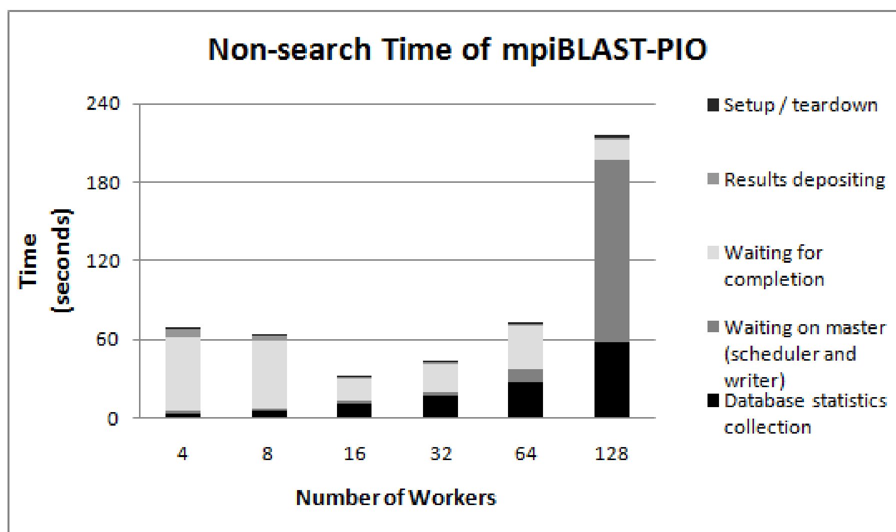

Figure 4.12: Breakdown of non-search related time in mpiBLAST-PIO.

to the total time.

The "results depositing" phase involves workers formatting and sending results to the master. Workers do this periodically during execution and send the data asynchronously. This allows the master to collect information regarding the final results throughout its execution, and allows workers to continue other work while results are being sent. Since workers process the results locally and send them to the master periodically, this phase represents only a small portion of the non-search time.

The "waiting for completion" phase is a noteworthy contributer to parallelization overhead, as it measures the time workers spend waiting on other workers to finish. Since the query sets and databases are simply partitioned by size, no predictions can be made about how long a given query takes to search against a fragment. If the query matches many of the fragment sequences very closely, searching will take longer than if the sequences were very distant. Thus, mpiBLAST-PIO provides the capability for workers to copy a new fragment to their local storage, so they can assist other workers in searching their database fragments.

However, these differences in search times are typically minor, so slower workers are usually searching their last query chunk against their database fragment at the time when other workers finish. In this scenario, the faster workers have no additional work to conduct, as

Table 4.1: The number of idle messages received by the master in mpiBLAST-PIO.

Number of Workers	Number of Idle Messages
4	2343
8	3997
16	7430
32	14814
64	30000
128	56710

all pending combinations of query chunks and database fragments are being worked on by other workers. Therefore, the faster workers must wait as the slower workers finish their assignments.

While query segmentation was implemented to help alleviate this issue by creating finer data decompositions, some cases show that the division is still too coarse. On average, workers in the 4-worker and 8-worker groups spend almost a minute waiting on other workers to finish searching their data sets. Some variation is seen between the other groups, which is due to happenstance matchings or mismatchings of data, which result in either relatively consistent search times (128-worker case) or varying search times (the 64-worker case).

The most significant source of overhead in the largest number of workers, and a major bottleneck for scalability, is seen in the "waiting on master" phase. The master, which handles both scheduling and writing, waits for incoming messages from workers. Upon the arrival of a message, the master determines the message's type and passes it along to the appropriate handler (either the scheduler or the writer). The scheduler determines what the worker should do next and notifies it with a message; the writer unpacks received results and writes them to disk. Once processing of the message is complete, the next message is handled. Waiting on these two handlers constitutes the entire "waiting on master" phase.

Since workers send an idle message to the master once they finish their current task, they must wait for a response from the master's scheduler before conducting more work. Further, since the master writes the worker's results immediately, the worker must wait until writing is finished before its idle message can be processed by the scheduler. However, since most writing occurs while the majority of workers are searching, only a small amount of time is wasted when a worker reports its results as it waits for the master to complete.

The second contributer to this phase is time spent waiting on the scheduler. Up to the 64 worker case, scheduling time is fairly minimal; however, with 128 workers, a large increase in scheduling activity is observed. The increase is largely due to additional communication between workers and the master, resulting in the master being flooded by idle messages from workers (Table 4.1). Severe query set segmentation results in workers executing single queries against a fragment (rather than multiple queries at a time). A finer granularity implies more frequent completions by workers; hence, workers request data from the master more often.

Each time the master receives an idle message from a worker, it spends time queueing the worker in order to serialize access to common data, constructing the next assignment, and notifying the worker of its next job. These steps take on the order of a few milliseconds to complete, but accumulate significantly when the master is bombarded by messages. Further, worker queueing can cause latency in the response time of the master.

With many small query sets, consistent search times can be achieved at the cost of a high scheduling overhead (i.e., the 128-worker case). With larger query sets, the master needs to assign work less often, so scheduling overhead is low; however, since the granularity of data partitioning is coarse, some workers might finish before others and must wait for their peers to complete (i.e., the 4-worker case). The goal is to provide a balance between scheduling overhead and consistent search times. This issue is discussed further in the potential improvements section.

Finally, the "database statistics collection" phase contributes to the overhead, especially as the number of workers grows. *This phase allows mpiBLAST-PIO to generate correct sequences and e-value scores, whereas mpiBLAST-1.2 produces approximate scores.* The increase in processing time is due to the overhead associated with collecting statistics from more heavily fragmented databases. Section 4.7 reflects on the importance of this issue and considers a potential solution.

4.6 Version Comparison

Comparing the performance of these two versions can help identify the effectiveness of changes made in the design of mpiBLAST and can lead to quantitative insights on the effectiveness of particular modifications between versions. Thus, this section will focus on a

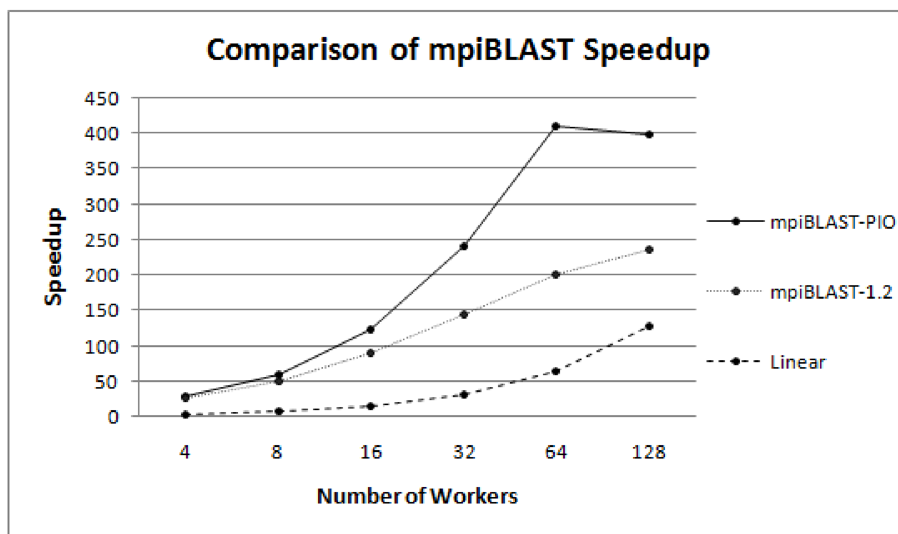

Figure 4.13: Speedup comparison of mpiBLAST-1.2 and mpiBLAST-PIO.

high-level comparison, in addition to covering evolutionary points of interest and discussing whether or not these changes are warranted.

Though mpiBLAST-PIO and mpiBLAST-1.2 see similar speedup with 4 workers (Figure 4.13), mpiBLAST-PIO quickly shows its superiority when used with larger numbers of nodes. With 64 workers, mpiBLAST-PIO exhibits twice the speedup that 1.2 does; however, the gap is shrunk with 128 workers, as PIO's speedup drops and 1.2's continues to increase. Note that both versions see super-linear speedup.

Further, mpiBLAST-PIO makes more efficient use of its workers during all executions (Figure 4.14), and even doubles mpiBLAST-1.2's efficiency at 64 workers. However, the drop in efficiency from 64 workers to 128 is fairly severe with mpiBLAST-PIO. Note again that both versions see efficiencies greater than 1.0 with all numbers of workers.

As expected, the two versions spend approximately the same amount of time in the BLAST search itself. Because the search algorithm for mpiBLAST-1.2 and mpiBLAST-PIO is identical, the search time is effectively the same. Thus the performance differences between mpiBLAST-1.2 and mpiBLAST-PIO must be attributed to the non-search time.

Figure 4.15 compares the non-search time of the two versions. While mpiBLAST-1.2's times stay relatively consistent, the non-search time in mpiBLAST-PIO almost doubles from 32

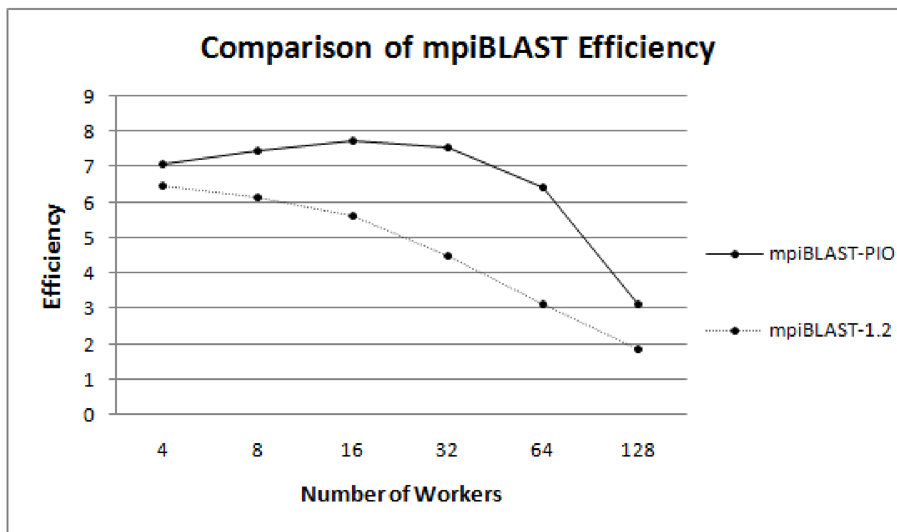

Figure 4.14: Efficiency comparison of mpiBLAST-1.2 and mpiBLAST-PIO.

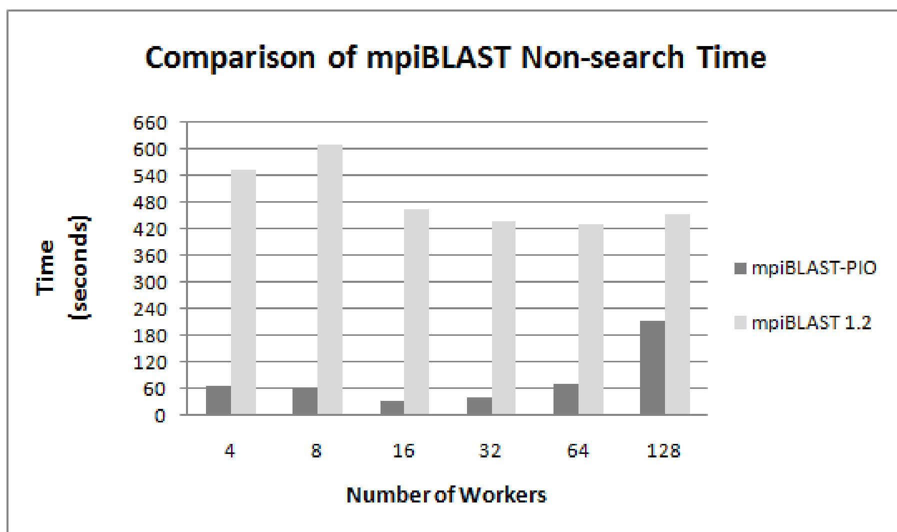

Figure 4.15: Comparison of non-search related time in mpiBLAST-1.2 and mpiBLAST-PIO.

to 64 workers, and triples from 64 to 128. However, mpiBLAST-PIO times remain less than those of 1.2, reaching at most half of 1.2's with 128 workers.

The mpiBLAST-1.2 non-search times are nearly constant across a large number of workers (i.e., 16, 32, 64, and 128) while PIO's increase, thus mpiBLAST-1.2 would likely make better use of more workers. The efficiency comparison graph (Figure 4.14) helps to visualize this, as the efficiency of mpiBLAST-PIO takes a sudden dip after 64 workers and nears that of mpiBLAST-1.2.

The database analysis phase of mpiBLAST-PIO is missing in mpiBLAST-1.2. Though this process adds extra time to PIO's execution, it ensures accurate results. Since mpiBLAST-1.2 does not perform this step, it provides less accurate results by approximating scores.

Both versions have workers that, at some point, wait on other workers. Since the distribution of sequences in the database is unordered, and since mpiBLAST-PIO partitions the query set, differences in waiting times can be explained by varying divisions of the data. With a certain number of workers, the decomposition of data may be even (in regards to execution time), while with other numbers that may not be the case. However, in all executions, worker waiting time is minimal and contributes only a minute to the overall time at most.

Recall that mpiBLAST-PIO incorporates query segmentation to partition the query file more effectively. In these experiments, the number of database fragments created equals the number of workers, and each worker starts with one fragment. As discussed in chapter 2, if a worker finishes searching the entire query set against its database fragment, it can take on the fragment of another worker to help even out search execution times.

However, query segmentation does not significantly help stabilize worker execution times (i.e., reduce worker waiting) in the tested configuration. Workers simply end up waiting on a worker that is conducting the search of its last query against its database fragment. Waiting workers cannot help complete the final query, as they can only copy the same database fragment to local storage and work on the remaining query themselves (which is exactly what the final worker is doing). Thus, query segmentation is better suited for environments where the number of total database fragments is greater than the number of workers.

Other tasks that both versions must complete include results gathering and writing, though they take different approaches in accomplishing them. mpiBLAST-1.2 workers simply collect all results locally, then forward the requested data to the master to be processed once the entire search is complete. On the other hand, mpiBLAST-PIO workers process their results

locally and forward them to the master periodically.

Considering that the phase labeled "waiting on master" of mpiBLAST-PIO (Figure 4.12) represents time spent waiting on the writer *and* on the scheduler, it is obvious that PIO outperforms mpiBLAST-1.2 in processing and writing results. Workers in mpiBLAST-PIO seem to spend very little time transferring results to the master because they do it asynchronously and periodically (hence the reduced time for "results depositing" with mpiBLAST-PIO). This way, by the time the entire search is finished, most of the results data has already been formatted and sent to the master. In 1.2, workers can only send their results to the master once the search is finished, and even then they have to be formatted (in serial) at the master.

In summary, mpiBLAST-PIO sees higher speedup and efficiency over all runs, due mainly to its improved handling of results collection and output. Since more work is done at the workers than with mpiBLAST-1.2, mpiBLAST-PIO parallelizes better. And, though the database statistics collection adds time to PIO's execution, it is necessary for more accurate results and does not push its execution time past that of 1.2.

However, some consideration should be given to how scheduling is conducted, as mpiBLAST-PIO's master experiences load with large numbers of workers. Further, certain bottlenecks may arise at the master from inefficient handling of requests from workers. These issues are discussed further in the following section.

4.7 Potential Improvements

It is clear that mpiBLAST provides a considerable performance advantage over sequential BLAST and that, for the most part, mpiBLAST-PIO improves upon mpiBLAST-1.2 by employing various techniques to improve efficiency. However, the mpiBLAST project can likely benefit from additional modifications (i.e., to mpiBLAST-PIO) which address the bottlenecks observed during experimentation.

4.7.1 Parallelized Collection of Database Statistics

As mentioned previously, the database statistics collection step is necessary in providing correct scores for sequence alignments. The amount of time attributed toward this task

increases with the number of nodes, and the process occurs at the master before any searching at the workers takes place. Hence, time is wasted as workers just wait at the beginning of mpiBLAST's execution.

Interestingly, the interface that the NCBI toolkit presents for statistics collection looks very similar to the one used for searching. Instead of telling the search functionality to perform a full-fledged search on the database, a flag is provided, telling the toolkit to scan the database and collect statistics. Further, mpiBLAST shows that searching can be conducted in parallel and that the results can be merged by the master.

Therefore, in an effort to efficiently use resources, the workers could retrieve database fragments (exactly as they do for searching) and collect statistics on their particular pieces. Then, they would report back to the master, where the data would be merged and broadcast out to the workers. In addition, workers would then already have local database fragments to search after receiving the adjustments from the master.

4.7.2 Task Threading at the Master

Another issue in mpiBLAST-PIO is the serialization of worker requests at the master. After workers complete their current task, they prompt the master's scheduler for a new assignment. Upon receiving the request, the scheduler decides what the worker should do next, and informs it via a return message. Further, workers may also report results to master, which are handled by the master's writer functionality.

This scenario can likely benefit from multiple changes, though these changes carry a common theme. First, since the scheduler typically knows what a particular worker's next task will be, it can process this information in a separate thread, while waiting for more worker requests. Since the scheduler often spends most of its time waiting for requests, it can use this time to determine the workers' next tasks. Thus, on the arrival of a request for work, the scheduler can respond immediately with the next task.

The second improvement involves separating the scheduler and the writer into separate threads. Since scheduling- and writing-related messages are handled sequentially, workers must sometimes wait after submitting their results to the writer before receiving their next task from the scheduler. If the master's scheduler and writer were run in separate threads, workers could request more work from the scheduler while the writer is handling results.

4.7.3 Adaptive Decomposition

In an attempt to reduce differences in execution times of workers (especially with large numbers of nodes), mpiBLAST-PIO splits the query set at a fine grain. This allows workers to compute on small sets of data and request more once they are finished to ensure a relatively fair workload for all workers. Unfortunately, due to the time associated with finding and packaging jobs for workers, which leads to the queuing of worker requests at the master, this process can introduce significant amounts of overhead.

This scenario can be seen in the 128 worker executions. Workers received database fragments roughly equal to the total database size divided by the number of workers. However, they received query fragments consisting of a single query. It is unlikely that using a slightly larger query chunk size would greatly affect the differences in search times between workers. On the other hand, larger query chunks would certainly decrease the number of requests for data sent by workers, therefore reducing load on the master.

A more general method for reducing scheduling induced load involves adapting the size of work chunks at runtime. If the master recognizes that it is receiving many requests from the same worker in quick succession, the master could send the worker more data to work on per request. Further, additional data could be recorded (such as remaining queries, average time to compute per query, etc.) to help facilitate decision making regarding query chunk size.

If not implemented carefully, such a solution could lead to increased overhead itself.

4.8 Conclusion

mpiBLAST-1.2 and mpiBLAST-PIO parallelize a specific sequence-search application, BLAST, to improve its speed. By distributing data to multiple nodes in a cluster, both versions are able to obtain super-linear speedup over NCBI's sequential BLAST. And, through multiple algorithmic improvements, mpiBLAST-PIO completes even faster than mpiBLAST-1.2.

The presented analysis indicates that both versions spend a significant amount of time searching and that, in general, time spent conducting non-search related tasks is minimal. mpiBLAST-PIO's performance makes it evident that non-search time can be reduced by conducting output processing in parallel, and that having a separate entity collect results

periodically is beneficial. Worker efficiencies can be improved by partitioning the search space through database and query segmentation; however, precautions must be taken, as a fine fragmentation granularity can introduce scheduling-related overhead.

mpiBLAST meets its goal in making NCBI BLAST searches considerably faster. But, just as previous algorithms and heuristic strategies evolved into BLAST, BLAST itself will likely evolve into something better. While researchers improve the performance of applications through parallelization, other researchers are improving the underlying search algorithms themselves.

The rest of this paper concentrates on developing a strategy to parallelize these constantly evolving applications without the need to reimplement the layers necessary for parallelization.

Chapter 5

Supporting Parallel Sequence Searches in Heterogeneous Environments

5.1 Introduction

The work that has gone into mpiBLAST marks one stage of the evolution of sequence searching as a whole. mpiBLAST exemplifies the power behind conducting BLAST searches in parallel, which makes a strong case for performing other types of sequence searches (and even other searches in general) over distributed resources. Further, the presented analysis shows that mpiBLAST is mostly limited by time spent searching and that parallelization overhead is minimal.

Unfortunately, though better sequence comparison algorithms are sure to be realized, the mpiBLAST parallelization infrastructure cannot support them. Even new releases of the BLAST toolkit find it difficult to fit into previous mpiBLAST molds, as the interface is specific to old libraries. If one desires to make new searches parallel, *ideas* can certainly be taken from mpiBLAST, but they must be used to construct a new foundation, as mpiBLAST's is far from generic.

Further, mpiBLAST enforces requirements on its operating environment, as it needs a cluster with MPI to run. Even though many corporations have a plethora of desktop machines

configured on an internal network, they still require the structure of a cluster to reap the benefits that mpiBLAST provides. By tying mpiBLAST down to a specific environment, its usefulness is reduced.

Hence, in an attempt to address these two major issues, we have developed a *search framework* named gridRuby. It supports running many types of sequence searches over general purpose (in addition to dedicated) environments, which is especially valuable in making use of readily available resources.

5.2 Motivation

There exists a significant barrier to entry in the conversion of a sequential application to a distributed, grid-based application: the application writer must manage all the communication, the serialization, the bookkeeping–essentially, the skeleton and scaffolding of a distributed system–in an application-specific manner. Further, intersecting the code necessary to perform a scientific process with the cross-platform marshaling and management of resources necessary in a distributed environment is a difficult task.

Unfortunately, systems like Globus [17, 20] and Condor [50, 11] are too general to address these types of issues. While they provide an interface for implementing parallelization, users must be explicit about how entities within the system interact. Even with embarrassingly parallel scatter/gather algorithms, data distribution must be reimplemented per application.

A search framework can address the issue of extreme generality by implementing a fixed scheme of parallelization (e.g., scatter/gather). Users of such a framework then simply need to write their application in a particular way so that parallelization can occur implicitly, without having to scaffold a communication framework through library calls. Thus, many applications can be parallelized with a search framework as long as they adhere to the scatter/gather paradigm.

In addition, the idea behind producing a grid-based framework around an existing application is not new. In regards to BLAST, it can be seen in the work conducted by Gardner, et. al, which runs mpiBLAST on an ad-hoc grid [18]. In this instance, mpiBLAST parallelized a sequential application with MPI to run on a cluster, while the additional glue code (implemented in a Perl script) brought together different instances of mpiBLAST running on

compute clusters all across the country.

However, this work is an ad-hoc solution that is hardly reusable; others interested in achieving the same task but in a different environment would have to re-implement much of that work in order to achieve similar results. Further, it duplicates functionality by requiring two different layers of parallelization (the cluster-based parallelization layer of mpiBLAST and the light glue code that brought together the different cluster systems into a grid).

Thus, offering parallelization in a separate scaffolding is a useful service, as it can handle the tasks needed to distribute an application onto a grid automatically. Further, it can do this in a manner that is agnostic of the application it is parallelizing, allowing it to apply to whatever task needs to be performed without having to know what that task is. This alleviates certain software requirements and associated hardships, such as installing, configuring, and maintaining MPI on multiple systems. It also requires minimal configuration, as such a framework could accept the arrival or departure of computational resources on the fly.

5.3 Conventions

The following conventions are used in the remaining sections:

- The *client* is the implementor of the code to be parallelized. Client code is, similarly, the code which is to be parallelized. BLAST is an example of potential client code.
- The *user* is the end-user of the parallel code. The user might be the client, or might be someone for which the application is being implemented. Specific to BLAST, an example user is one who will conduct parallel BLAST searches with the final parallelized client code.
- When this work is referred to as a *framework*, it is not to imply that the work represents an API. In no way is the framework something which is programmatically invoked from client code as if it were a library. Instead, it is a framework in the sense that it builds a frame around client code which adds additional functionality, while keeping the client code mostly intact.
- As with most distributed systems, a *node* is a single entity, or actor, in the system. In this framework, a node is, in addition to being a generic node, one of a few specific

types (node types are described in Section 5.4.2). As a point of reference, mpiBLAST has nodes of type master and of type worker.

- An *arbitrary grid* is a collection of nodes which may run different operating systems on different architectures with different security policies. This collection may be modified at any time, with nodes being added to or removed from the system in an unpredictable fashion. A dynamic corporate network is an example of this, with various devices connecting (i.e., a laptop joining the wireless network) and disconnecting (i.e., a desktop machine rebooting) over time.

5.4 The Framework

The gridRuby framework provides an frame for an application or library, offering distribution services. It is written in the Ruby programming language, which is used as a cross-platform common ground and a shared runtime environment from which client code (written in a language more common in high-performance computing, such as C) can be invoked. The framework handles all the tracking, message passing, and division of labor tasks, effectively causing the n instances of the underlying client code to believe that each is an independent entity working on a single problem of size p/n (rather than using one instance to work on a problem of size p).

The efficient and automatic parallelization of arbitrary code is a difficult problem, and gridRuby certainly does not solve it. Instead, the taken approach is less all-inclusive: if an application follows a certain paradigm and maintains a high ratio of computation to communication, the framework will provide low-effort parallelization on an arbitrary grid. It conforms to a common paradigm, the traditional scatter-work-gather sequence seen in many parallel applications. Since this pattern is common in computationally-expensive scientific applications, it is an acceptable requirement.

In the case of library code, where functions for defining the search space and performing the search are exposed as methods, parallelization should be extremely easy and does not even require familiarity with the code to be parallelized, only its public interface. Even in the case where the client code is written in a very sequential manner as a self-contained application, parallelization using this framework is easier than writing all the same code by hand, as it is only necessary to provide ways to access and load relevant internal state from the outside.

An additional restriction imposed on the client code is that workers must be independent of one another. Workers can, however, communicate with a master and database node, allowing for indirect communication between workers (thus providing for aggregate computation scenarios). This framework will not provide explicit support for the intercommunication necessary for heavily-interconnected applications like Barnes-Hut [4].

These restrictions are similar to those found in the MapReduce [14] programming model, which is designed to efficiently parallelize algorithms by applying similar transformations to each element (using parallel processing) and aggregating the results. By enforcing independence of elements and a common paradigm for performing computation, a broad category of sequential applications can be supported without aspiring to automatically parallelize *everything*.

5.4.1 Justification of Ruby Language

To say that the Ruby scripting language is rarely seen in supercomputing is an understatement. While use of FORTRAN, C, or even C++ might pass without comment, using a language from the other end of the performance spectrum requires justification.

Ruby provides an environment for rapid prototyping, maintainable development, and clear, concise coding. As a scripting language, Ruby places programmer productivity high on its list of goals. It also includes higher-level structures such as string, associative arrays via hash tables, regular expressions, and even anonymous functions as first-class language features instead of tacked-on libraries. And since it is an interpreted, dynamic, and reflective language, all classes (including system library classes) can be modified at any time.

However, these features come at the price of performance. A Ruby grid framework, or a grid framework written in *any* language, would never see acceptance if it did not offer decent performance.

To alleviate this issue, gridRuby makes use of specific cross-language communication facilities. Ruby provides an API which can be used in other languages, especially C and C++, which makes communication between Ruby and these languages straightforward. While the Ruby language is not mature enough to form the core of a high-performance application, choosing a slower language that allows greater programmer productivity and offers more useful features should not hinder the project.

As an example, parallelization in mpiBLAST, when handled correctly, contributes little to the overall execution time. Thus, as long as time-expensive parts of the code are handled by a high-performance language, and parallelization is handled by the framework, there should be little effect on performance.

In addition, the Ruby language maintainers plan on incorporating a high-performance virtual machine interpreter [45] into the main branch of Ruby development. While excellent results can be achieved using the standard Ruby interpreter, the fact that the language seems to be moving toward greater performance is additional support that Ruby can serve as a framework for parallelization.

5.4.2 Node Types

The framework's approach leverages Ruby's ability to integrate with other languages to provide a common framework across all OS platforms and hardware architectures that offer a Ruby implementation. Structural support is provided for a logical decomposition of problems into distinct steps, each of which accomplishes a piece of the entire problem. Furthermore, a robust toolkit is supplied to assist in the interfacing between the Ruby code and the client code, which in the ideal case requires none of the client code to be modified.

To decompose problems properly, the concept of nodes is maintained. Each node has a type, and this type determines the role of the node in the system. The types, which will be covered in detail below, are worker, database, querier, and tracker. Each type must have a corresponding portion of the client code which handles the tasks of that type, with the exception of the tracker. The tracker node is implemented entirely within the framework itself.

Note that for these sections, the descriptions are couched in the terminology of search applications, as most applications decomposed in this fashion can be expressed as search applications; they consist of a range of data which must be processed and an expression which must be evaluated against all data in that range. Other scatter/gather type applications, which are not strictly searches, can still be described in this fashion and can still be parallelized in the same way.

The Worker

The worker performs the actual computations in the system. As such, most systems will be comprised primarily of workers. Because of this, workers are also the nodes which initially see submitted jobs and which display the results of completed jobs.

The worker code could, in theory, simply be the search code from the sequential application without modification. However, for real applications, this is unlikely, as the interface (i.e., the function declaration) between the framework and the search code is relatively tightly defined. The needed information needs to be packed in a single data structure and unpacked on the other end, and then passed into the real search function from a compatibility layer (this is the same approach used by the pthreads library, among others). Optimally, this thin glue would be removed entirely, but it is necessary for cross language compatibility.

The Database

The database manages the search space, partitioning it into chunks which are sent to workers for processing. The database also manages the query, so if query segmentation is desired, it can be performed as well. The database is the portion which is least likely to be already implemented, as sequential applications do not generally do this sort of segmentation, as it is unnecessary.

In the worst case, the database may need to be entirely new code. However, this should be a rare event and minimal effort when necessary. The database is also a prime source for optimization, both for the client-code implementors and for future development of the framework, since it handles the division of the problem.

The Querier

The querier is, conceptually, the simplest of the nodes in the system. In addition to actually performing the computation, it is necessary to have some sort of interface for submitting a new job. Since the actual generation and gathering of data is application-specific, this is the job of the client code. This code should be very easy to integrate with the framework, since a sequential search also needs to perform this function, so the logic for this (which is already in place) should suffice for the querier.

The querier submits a job to a worker. This worker then becomes the "owner" of the job and gains the responsibility of aggregating the results. Though the querier could act as a repository for results, this leads to some concern regarding its availability. Since the querier functions as a simple interface (command-line based, graphical, or even web-based) for job submission, the user should be able to terminate this application without fear of losing data. With the worker acting as a gatherer in the background, the worry of keeping an interface to the system available is removed.

The Tracker

Initial interconnection designs involved each entity connecting to other relevant entities directly [1]. However, in this approach, the number of connections each entity has to maintain increases linearly with network size, making the total connections in the system increase quadratically. As more networks become connected to one-another, the number of inter-network connections also increases, requiring broad ranges of ports to be reserved locally and forwarded remotely. This can result in scalability issues for the network and the network maintainer, and creates enough of a security concern that deployment on a real network would no longer be viable.

Instead, a special type of node, called a tracker, is used. The tracker nodes function as routers for traffic, which greatly simplifies the network code in each other type of node. Each node must maintain only one connection, the one to its local tracker, which localizes all the communication complexity in one place (where it can be optimized for). Further, since trackers do not assist in computation but are completely devoted to ensuring efficient communication, the latency is less than it would be if, for example, a database had to balance the work of partitioning, assigning, and monitoring the computation with the task of managing the possibly large number of connections incoming from workers.

One valid criticism of this design is that the tracker becomes a potential bottleneck for the system. However, it should be noted that, of all the nodes, the tracker is the one doing the least amount of work. The number of connections through the tracker increases just linearly with number of nodes in the system in the worst case (and may be sub-linear, as a link to another network requires just a single connection, to that network's tracker). Since the tracker does not have to do any processing of data messages itself, the additional latency

[1] TCP Hole-punching [16] was also considered, but it is not supported across all platforms.

imposed by such a bottleneck is likely to be eclipsed by the processing time required by the worker nodes.

5.4.3 Integration

To integrate an application or library with this framework, the functionality provided by the application must be framed and exposed in approximately the way that the framework expects. To accomplish this, users describe the interface using YAML [49], a data description language that can be easily read and written. By using a YAML document template and by filling in function names, parameter information, and relevant composite type definitions, clients can provide enough information for the gridRuby auto-generator to produce glue code that is specific to their application (Figure 5.1).

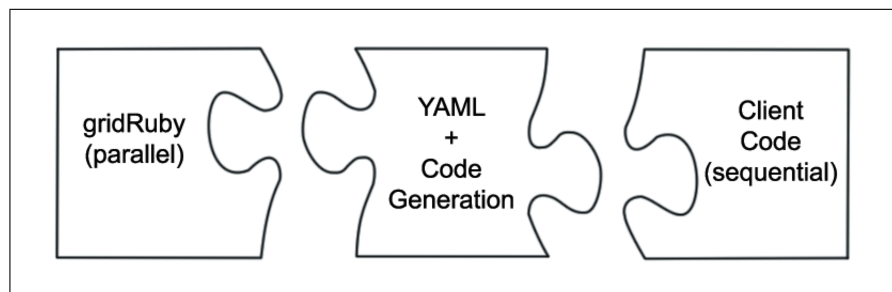

Figure 5.1: A YAML configuration and automatic code generation connect a previously sequential application to gridRuby.

In the optimal case, nothing more is necessary. If the client code does not expose its functionality in an appropriate way (i.e., the application lacks modularity) or if additional hand-tuning for performance is desired, more coding may be required. However, this work should be minimal, as the client side simply needs to provide an interface to functions that will be run on each type of node (i.e., database, worker, and querier).

5.5 An Example: BLAST Integration

To understand the process of integrating an existing application into the gridRuby framework, one must consider it in terms of a real application. Therefore, a particular BLAST search application, `blast2seq`, has been parallelized with the framework and will be used to facilitate the following discussion.

`blast2seq` is a sequential BLAST application released with the NCBI C++ toolkit. It is a command-line application that accepts as input, among other options, a type of BLAST search (dependent upon the type of database and query set), a query set filename, and a database filename. Both the query and database sequence data must be provided from plaintext files (unlike `blastall`, which uses a formatted database). This application was chosen because it already exists (that is, it was not produced for the framework as a contrived example), it represents a compute-intensive search, and it is written in a straightforward nature in C++.

5.5.1 Configuration

Though the gridRuby framework aims to be as versatile as possible, it cannot determine how to parallelize a sequential application without help. Some configuration is necessary at least to tell the framework what to interface with in the sequential code. Hence, a couple of steps are necessary to provide gridRuby with enough information to operate correctly.

Upon receiving the necessary command-line arguments, `blast2seq` reads in all query and database sequences, conducts a BLAST search on them, and serializes the results to NCBI's ASN.1 format, outputting either to the screen or to disk (depending on the options provided). Since the application implements this in a serial manner, it is necessary to divide the code into components which match the interface expected by gridRuby. `blast2seq` represents a case that involves implementing additional, though minimal, client-side code to adhere to the framework's expectations.

Assigning the sequential applications tasks to gridRuby nodes is straightforward. The querier node of gridRuby should read in the arguments to the program, as the querier deals with interacting with the user and submitting jobs. The database node should take care of storing sequence data and reading it in as necessary. And the worker node (or nodes) should actually

conduct the BLAST search. In addition, the ability to serialize and write results should be available. The transition toward modularity can be seen in Figure 5.2.

Since the functionality for the querier and the worker are already in place in the sequential code, they were dealt with first. Hence, all of the argument processing functionality of `blast2seq` was moved into a separate function named `cppGetArguments`, and all of the search functionality was placed into a function named `cppSearchChunk`. As mentioned earlier, results gathering is a separate task, but takes place at the worker that received the job submission. Therefore, the results writing steps were moved into a function called `cppWriteResults` (again, see Figure 5.2).

If this modified application was to be run sequentially, there would be no need for a database node; the querier could simply process the arguments, read in all sequence data, and send it to the remote worker. However, this provides no improvement over the sequential application (except for being able to submit a job to a remote node). Instead, to take advantage of data parallelism, the input data can be split into pieces that can be handed out to workers (much like mpiBLAST). The database node exists for this reason.

Unfortunately, the code necessary to split the data into chunks does not exist in the sequential application (because, again, there is no need for the decomposition of data). Therefore, the sequence reading functionality of `blast2seq` was modified and moved into a function named `cppGetChunk`. Rather than reading in the entirety of the sequence data to be searched on, this new function reads in a small chunk of data on each invocation. In this way, different pieces of data can be returned for each worker so that workers can compute on distinct data in parallel.

It is also necessary to inform the database of arguments to the program (so it knows which files it should be working with). Hence, the `cppSubmitArguments` function was introduced. It simply accepts arguments and opens the database file, so that future calls to `cppGetChunk` will succeed in reading data.

After these modifications, the code from the sequential `blast2seq` looked different, but is functionally very similar. In fact, the new functions just introduce modularity to the old sequential application and can even be used to re-implement it (Figure 5.3). Typically, as is the case with this example, the return value of one function becomes the input parameter of another.

Even though the application was split into separate components, the framework had no way

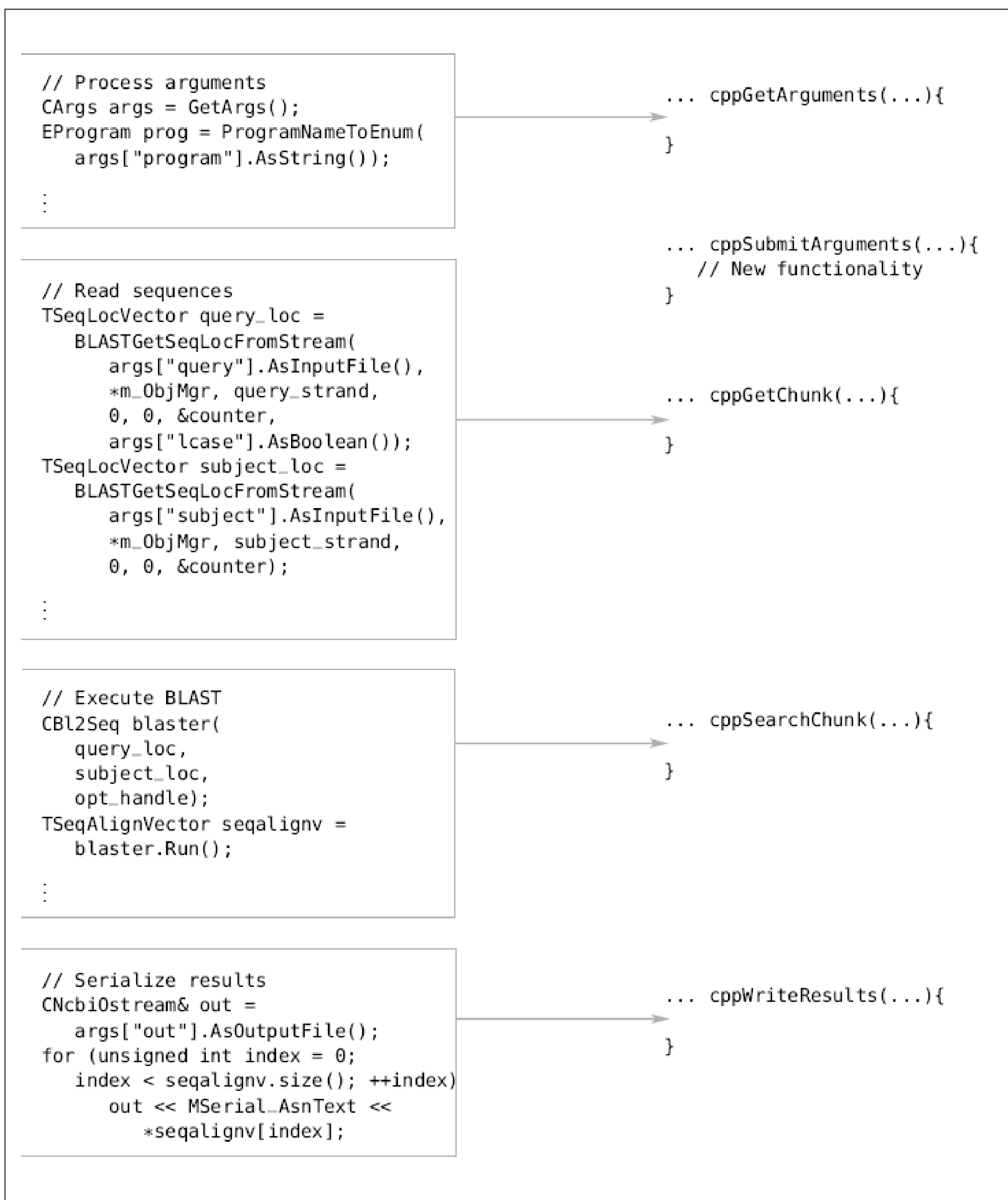

Figure 5.2: `blast2seq`'s sequential workflow and the modularization of its distinct tasks.

```
cppSubmitArguments(cppGetArguments());
while(chunks remain)
   cppWriteResults(cppSearchChunk(cppGetChunk()));
```

Figure 5.3: The hypothetical new workflow of `blast2seq`, which uses modularized functionality.

of knowing how to parallelize these pieces. As mentioned previously, gridRuby interprets a particular YAML document which helps describe the mapping between the client code and the interface it expects. Hence, a YAML document (Figure 5.4) was constructed for this purpose.

First, the "description" section provides information about the search, such as its name, its version, and the language in which it is written. These things help identify the search when compared to previously implemented versions, and helps the code generator to make decisions about what to name things and what language to use.

Second, all data structures used for parameters and for return values are defined in the "structs" section. This information is used by the code generator, so that it knows how to automatically serialize and deserialize data to and from the network. Currently, the framework only supports the passing of structures of basic types into and out of functions[2].

Finally, the "interface" sections define the mapping between the gridRuby interface and the client code functions. This step tells the code generator which entity (worker, database, or querier) each client-side function belongs to, where the function header can be found, the name of the function, and the types of parameters and return values the function will use.

It should be mentioned that producing the YAML document before making code changes might be beneficial, as it lists the functionality that is necessary for integration. The template can act as an "outline" for the client code to follow, acting as a simplified representation of the structure of the code.

[2]Unfortunately, a tool like SWIG [54] will not remedy this issue, as it does not automatically produce serializable Ruby objects.

```yaml
---
Description:
  getType: BlastExtension
  getVersion: 1
  extensionLanguage: c++

---
Structs:
  out: {results: string}
  arguments: {data: string}
  chunk: {queries: string, subjects: string}

---
QuerierInterface:
  getJob:
    Include: Querier.hpp
    Name: cppGetArguments
    Param:
    Return: arguments*

---
DatabaseInterface:
  newJob:
    Include: Database.hpp
    Name: cppSubmitArguments
    Param: arguments*
    Return:
  getChunk:
    Include: Database.hpp
    Name: cppGetChunk
    Param:
    Return: chunk*

---
SearchInterface:
  search:
    Include: Search.hpp
    Name: cppSearchChunk
    Param: chunk*
    Return: out*

---
ResultsInterface:
  gotResults:
    Include: Results.hpp
    Name: cppWriteResults
    Param: out*
    Return:
```

Figure 5.4: The YAML configuration file used for integration of BLAST with gridRuby. Template text is in gray, while text provided by the user is in black.

5.5.2 Adhesion

Following the initial configuration, the client code needs to be exposed in a way that the gridRuby framework expects and understands. Fortunately, the above mentioned YAML configuration file and code generation are used to resolve this issue automatically.

Given the above document, the gridRuby parser first determines that it should use C++ for any generated code (currently, the framework supports C and C++). Then, it collects information regarding the C++ structure types that will be used as parameters and return values. As an example, one such type described in the YAML is named `out`, which contains a single member, `results`, of type string.

Then, for each interface definition (e.g., QuerierInterface), a Ruby class is produced automatically. These classes are defined in separate C++ files through an API provided by a Ruby header. Further, class methods are generated, which carry out framework specific functionality (namely, serializing and deserializing data) and call the corresponding user specified function. In this way, the Ruby end of the framework knows to call, for example, `getJob` in the `QuerierInterface` which will end up calling the client function `cppGetArguments`, after taking care of the necessary data conversions.

After the glue code is produced, it, along with the client-side functions, are compiled into a single shared object. This process is facilitated by built-in Ruby functionality which creates an appropriate makefile for the project. The resulting shared object can be included by the gridRuby framework, and the Ruby classes defined in the glue code can be used directly.

5.5.3 Execution

The scenario involving a single gridRuby worker flows much like the theoretical, re-written sequential application. However, additional steps are required to marshal C++ data to Ruby on one end, and then to convert the resulting Ruby data back to C++ data on the other end. Though all of this occurs automatically, the process is described here to provide a deeper understanding of the framework.

A typical environment would likely have each actor on a different system, though this is not strictly necessary. Further, the order in which nodes are started does not matter, but all entities should be started before a job is submitted. Database and worker nodes will connect

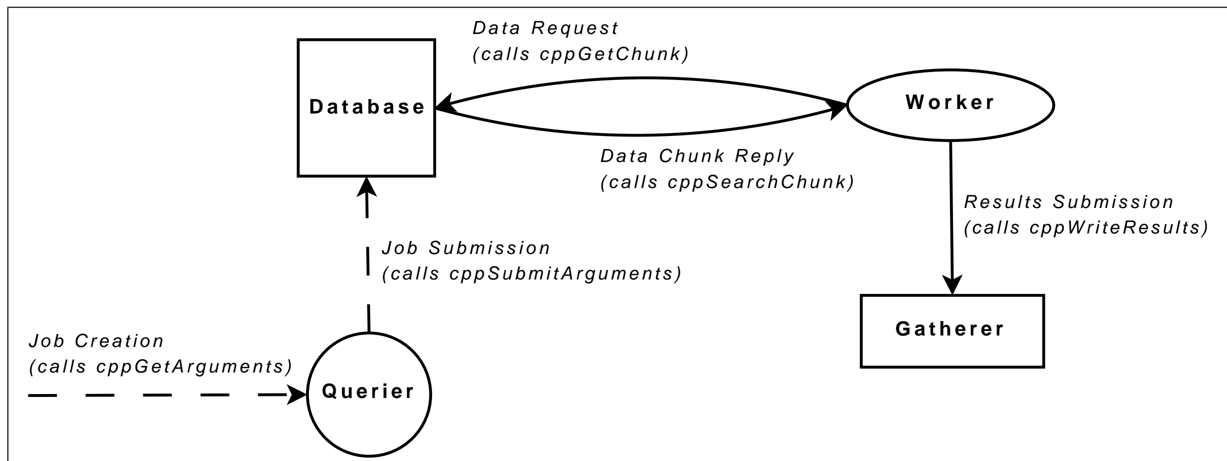

Figure 5.5: The process of calling into client code. Dashed lines represent actions that happen only once per job, while solid lines represent actions that may occur multiple times per job.

to their tracker once notified of its presence. Figure 5.5 shows the sequence of client code calls.

Once started, the querier node simply calls into the appropriate client-side function; in this case, that function is `cppGetArguments`. The Ruby querier accepts command-line arguments, which can then be passed to the client function. Further, since the client-side code is executed as it normally would be, it can do any number of things (like fail due to invalid arguments, prompt for more information, etc.). However, the `cppGetArguments` function simply accepts the same command line arguments as the original application and verifies these arguments before returning them to the framework.

When the client function completes, the return data is converted into a Ruby format and serialized for transfer over the network. The framework forwards these data to the database node, which deserializes the Ruby data into the appropriate, local C++ data structure. This structure is then passed into the `cppSubmitArguments` function, which reads the arguments and opens sequence files. The database node then waits for chunk requests from workers and notifies the tracker that it is ready for incoming data requests.

The tracker then notifies each of the connected workers that a new job is available. Each worker prompts the database for work, which results in a call to `cppGetChunk` for each

request. This function returns a structure containing the entire query set and a part of the database (the number sequences to read from the database is provided as a constant). Again, the glue code wraps these data up and the framework ships them off to the requesting worker.

Back at the worker, the data is converted and passed into `cppSearchChunk`. The query sequences are compared to those of the database chunk, and results are produced, serialized to ASN.1, and returned. The resulting structure is converted for transfer over the network and is passed to the results gatherer (the originating worker). There, the results are converted back to their C++ format, passed into the `cppWriteResults` function, and are output to file.

Once workers pass their results along, they immediately request more work from the database node. Workers continue to search on chunks until no more data is available.

5.6 Results

Though the transformations described above will yield a functional parallel application, a few simple changes can make a world of difference in performance. Since the application was originally written to perform in a sequential manner, NCBI objects (for storage, searching, etc.) that would normally be created only once are created multiple times in the parallel application (on each invocation of a client code function). This leads to a rather drastic degradation in performance.

The simple and obvious solution to this problem is to pull out the declarations of the reusable objects, and register them as globals. Then, these objects can be created once and reused throughout the entire execution. This leads to substantial performance benefits especially in the case of the BLAST search object, as its high initialization cost is incurred only once.

While an in-depth performance analysis (like the one from chapter 4) was not conducted on gridRuby, a few simple tests were run to get an idea of how well it performs with an integrated search. Experimentation consisted of running the `blast2seq` application as a baseline for the sequential case, and then running gridRuby (with the integrated `blast2seq` code) over 10 nodes of a cluster.

In the gridRuby environment, a single database node held a query file containing a single

Table 5.1: The speedup and efficiency of `blast2seq` when integrated with gridRuby and run over 8 workers.

	Runtime	Speedup	Efficiency
blast2seq	23288.41	-	-
gridRuby + blast2seq	620.9679	37.5034	4.6879

query and the GenBank `nr` database (approximately 2.7 gigabytes in size). The tracker was started on a separate node, while 8 workers were assigned to their own separate nodes. After all entities automatically discovered the tracker, a job was submitted from one of the workers, and the search ran to completion.

Even though gridRuby was designed with grids, not clusters, in mind, it still shows impressive speedup over the sequential application. gridRuby saw over a 35-fold speedup with only 8 workers, yielding an efficiency of over 4.5 per worker (Table 5.1). As is the case with mpiBLAST, paging of the database (as it does not fit into the memory of one node) makes for poor performance of the sequential application, which allows gridRuby to achieve super-linear speedup.

5.7 Related Work

gridRuby spans the work of multiple areas including minimal effort parallelization, remote job submission, and grid-based BLAST implementation. Major players in these areas are discussed below.

The Globus toolkit [17, 20] is comprised of a collection of services and libraries that facilitate grid computing. It implements many grid-related standards and provides a rather expansive API for conducting work over grids. The toolkit saw its first released in 1997 and has since grown into a very large solution that many consider to be the "de facto standard" in grid computing.

Unfortunately, when coupled with an application, such a large solution to address the issue of grid computing can complicate the tasks necessary for parallelization, especially when the target application is "embarrassingly parallel." Further, it requires client code to know that it will not be run in isolation (hence, it must be written as a parallel application), and

requires the configuration and maintenance of client machines.

Condor [50, 11] is a batch system that can use dedicated resources or idle machines to run submitted jobs. It provides job check-pointing and migration, so that jobs can be transferred from active machines to those that are idle. Further, Condor supports sequential, MPI-based, and grid-based execution environments.

While Condor provides support for MPI as well as its own master-worker library, these strategies are better suited for a cluster environment. Further, to submit jobs to a grid, Condor requires a separate system (such as Globus) to function. Though this is useful, building the infrastructure to support a grid-based job is involved, and still requires that the target application be written in an explicitly parallel style.

Pastry [44, 10] provides intelligent routing over a structured peer-to-peer protocol. The system addresses many network-related issues including node failure and recovery, routing and routing distances, and scalability. While these strategies would prove useful in grid-based sequence search applications, many such applications require multiple types of nodes, with *different* responsibilities, to accomplish the search. Self-organization is merely a subset of the node coordination problem, as sequence data needs to be broadcast to and results need to be gathered from workers.

It should be noted, however, that much of the work conducted by Pastry (especially fault tolerance) is complementary to gridRuby's current capabilities, as work on gridRuby was focused toward target applications rather than the network. Incorporation of these ideas into the gridRuby system would create a more robust sequence searching framework.

The MapReduce programming model [14] is very similar to gridRuby in that it provides distribution services by calling into user functionality and without the user explicitly designing parallelism. gridRuby uses more node types than MapReduce to provide the capability to distribute load across multiple machines (e.g., through use of separate database and tracker nodes) and to provide an interface to the system (i.e., through the querier). gridRuby focuses on the ease of target application integration, while MapReduce incorporates fault tolerance and job checkpointing. As is the case with Pastry, gridRuby could make use of these techniques to improve reliability.

Multiple grid-based BLAST implementations exist which use these systems. In fact, mpiBLAST-G2 [37] is a modified version of mpiBLAST which uses the Globus toolkit and MPICH-G2 [27] for parallelization over grids.

Another application, GridBLAST [31], also uses the Globus toolkit to run BLAST on grids, but the code does not stem from mpiBLAST. GridBLAST is comprised of a set of Perl scripts that invoke the BLAST executable on multiple nodes, with each node searching a subset of the queries against the entire database. Unfortunately, the application requires PBS for use with clusters and does not seem particularly robust.

PackageBLAST [48] takes a different approach to distributing tasks across a grid. Instead of always distributing workloads of fixed size to computational nodes, it incorporates multiple schemes for adjusting the distribution of work. Static and self-scheduling strategies are provided, in addition to support for user defined schemes. PackageBLAST relies on the Globus toolkit as well.

5.8 Conclusion

The gridRuby framework addresses two major issues with traditional sequence search parallelization. Such traditional applications are usually specific to a particular type of search and they rely on dedicated environments to run. Those that can execute in a grid environment require substantial configuration and support.

gridRuby handles these issues by acting as a generic framework for sequence searches to fit into. By implementing target applications to use the common scatter/gather paradigm, implementors can easily drop their applications into the framework. Further, much of the traditional configuration is unnecessary, as nodes discover one another and create connections on the fly. Cross platform issues are resolved by using the Ruby language, which acts as a common ground between environments.

Of particular importance is the fact that clients can implement their searches in high-performance languages like C and C++. In this way, gridRuby is able to pull resources into a grid through the Ruby based framework, while performing computationally expensive sections in a high-performance language. Thus, gridRuby can achieve super-linear speedup with an integrated application, while making parallelization of that application hassle-free.

Chapter 6

Conclusion

6.1 Future Work

There exists a broad area in which future work can be conducted. mpiBLAST can benefit from further experimentation through variations of the experimentation variables described in 4.3 (e.g., using different systems with different numbers of nodes, using different toolkits or different searches). Such experimentation may lead to the realization of certain problems or bottlenecks as the presented analysis has, which can motivate changes to mpiBLAST's algorithm. The issues identified and solutions provided in 4.7 are likely just a subset of those to be discovered.

In the case of gridRuby, even more possibilities exist. Since the framework is still in its infancy, it can benefit from incredible amounts of change. Such improvements can be divided mostly into three areas: robustness, features, and measurement.

gridRuby could be made more scalable and *robust* through further implementation at the framework level. Possibilities include support for:

- Direct, node-to-node communication for internal networks
- Random entrance and departure of nodes during searches
- Advanced failure handling through client-code
- Database node replication

gridRuby conducts all communication, including that of the local network, through the tracker. While this reduces complexity, it is not strictly necessary, and creating direct connections between local nodes would likely improve speed on local networks. In addition, fault tolerance could be improved by allowing workers to join and leave searches during their execution. Currently, gridRuby expects that every work chunk will be finished to completion (thus, it does not account for node failure scenarios), and it does not allow workers to join a search after it has been started. Reliability and performance could be further improved through support for node replication (e.g., support for multiple database nodes).

Many *features* could be added to gridRuby to make it a more appealing solution. These include:

- Authentication and group management
- Encryption
- Web-based interface for job submission and interaction
- On-the-fly job size variability

Security policies are important to many jobs, as the correctness of data is essential. Further, authentication and security of data are important parts of many protocols, and gridRuby is no exception. Currently, gridRuby expects its entities to interact reliably and does not account for malicious activity. It could support restrictive joining through authentication, so only trusted machines could interact in searches. In addition, many companies may wish to keep their searches and results private, which is of major concern when these data are being shipped across administrative boundaries and huge geographic distances. Of course, support for encryption would help to solve this problem.

To provide metrics for *measuring* the effectiveness of gridRuby, the following areas could be explored:

- Usability studies
- Profiling and in-depth performance evaluation

By breaking the target application into specific tasks, gridRuby provides an outline for applications without imposing the restrictiveness of a parallelization library. This helps

facilitate application development and parallelization, but its ease of use has only been tested internal to this study by its developers (and even then informally). To gain an understanding of how easy or difficult gridRuby is to work with, usability studies are necessary. These studies could be conducted at both the client (implementor of the application) level and at the user (end user of the application) level. Client interaction could be measured in terms of length of development cycles (as compared to traditional parallelization methods) and complexity of code, while user interaction would focus on deployment and integration into existing environments.

In addition to these three categories, even more idealistic design changes could be made. One can imagine a system in which entities are even more advanced (e.g., workers contact each other as necessary for information, entities communicate with the closest nodes, nodes change types to facilitate faster searching, etc.), rather than following a fixed pattern (e.g., contact database for data, search on data, submit results, repeat).

Again, one should not feel restricted by the barriers of a particular language when parallelizing high-performance code. These supportive roles can be implemented in a language that easily addresses such concepts (even in a cross-platform way), as long as the time-intensive parts are handled in a high-performance language. gridRuby exists to show that these ideas are possible.

6.2 Reflection

This work looks at the methodologies and performance of the parallel sequence searching tool mpiBLAST. As a result of this analysis, a generic framework was introduced to show that a relatively slow language can make use of high-performance components to obtain impressive speedup with a BLAST search.

The intent of gridRuby is to inspire creativity while accomplishing work faster. Though implementations based on traditional parallelization libraries certainly perform well, they restrict possibilities by promoting hand-tuned reimplementations. Parallelization frameworks, on the other hand, promote quick solutions to the problem of parallelization. By interacting with a known working system of parallelization, clients can be creative with what they parallelize without having to worry about troubleshooting network or cross-platform issues.

Further, this strategy is not just a hack. Client code cleanly integrates into gridRuby, while the framework knows nothing about what the client code does. The idea of using a generic framework for parallelization is powerful and can be used to accomplish many types of tasks across many platforms. In fact, even though it was designed with sequence searching in mind, gridRuby does not need to be restricted to this area. Client code could be written to perform parallel greps, to execute commands against a remote SQL database, or to record the status of the machines on a network; there is a large number of possibilities.

6.3 Conclusion

Many important tools exist to help determine links between organisms and genes. BLAST is one such tool that is used by countless numbers of biologists daily. By noting similarities between known and newly sequenced genes with BLAST, one can quickly identify properties of new genes. This can prove especially valuable in drug creation or in dealing with a newly discovered virus, particularly when speed is a concern.

Parallel implementations of these sequence searches, like mpiBLAST, make this process even faster by using cluster resources to conduct BLAST searches in parallel. By providing an infrastructure around these searches, mpiBLAST is able to handle tasks like sorting and merging results automatically. Impressive performance is realized due to parallel searching and output processing, as well as from the efficient use of distributed memory.

Further, analysis of mpiBLAST's execution shows that searching occupies a large amount of the overall execution time and that parallelization overhead is typically minimal even with large numbers of nodes. This is likely the case with many sequence search algorithms, as many are particularly search intensive.

Though BLAST is important, there exist many other searches that are just as valuable, with more to come in the future. Further, these searches could take advantage of the same parallelization techniques that mpiBLAST employs to improve their speed. Unfortunately, the mpiBLAST parallelization framework is specific to the NCBI toolkit and must be heavily modified to support other sequence search algorithms. In addition, it requires a set of tightly coupled machines (usually realized as a cluster) to run.

Therefore a generic grid framework for sequence searching, named gridRuby, has been devel-

oped. After minimal modifications, an existing sequential application can be automatically parallelized by gridRuby. Since the framework expects the application to present its functionality in a typical scatter/gather style, many sequence search applications can be dropped into gridRuby with very little effort. Further, the parallelization code remains separate from the scientific code, allowing for modification of one without concern for the other.

And though gridRuby is not yet an optimal solution, it represents a different but formidable way of conducting computation in parallel. While much effort is spent in parallelizing applications by hand, the presented work shows that this is not always necessary. Further, as is shown by gridRuby's ability to achieve super-linear speedup, an adjustable, reusable solution's performance can compare to that of a hand-tailored, specific implementation for certain types of applications.

Thus, scientists can use the framework to obtain results from their application faster, without the difficulty of implementing a new parallelization strategy from scratch.

Bibliography

[1] S. F. Altschul. Amino acid substitution matrices from an information theoretic perspective. *J. Mol. Biol.*, 219(3):555–565, June 1991.

[2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J. Mol. Biol.*, 215(3):403–410, October 1990.

[3] S.F. Altschul, T.L. Madden, A.A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D.J. Lipman. Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic Acids Res.*, 25:3389–3402, 1997.

[4] J. Barnes and P. Hut. A Hierarchical O(NlogN) Force-Calculation Algorithm. *Nature*, 324:446–449, December 1986.

[5] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and D. L. Wheeler. Genbank. *Nucleic Acids Res*, 35(Database issue), January 2007.

[6] R. D. Bjornson, A. H. Sherman, S. B. Weston, N. Willard, and J. Wing. Turboblast(r): A parallel implementation of blast built on the turbohub. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 325, Washington, DC, USA, 2002. IEEE Computer Society.

[7] Paracel BLAST. http://www.paracel.com.

[8] M. Cameron, H.E. Williams, and A. Cannane. A deterministic finite automaton for faster protein hit detection in blast. *J. Comput. Biol.*, 13(4):965–78, 2006.

[9] Michael Cameron, Hugh E. Williams, and Adam Cannane. Improved gapped alignment in blast. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 1(3):116–129, 2004.

[10] M. Castro, P. Druschel, Y. Hu, and A. Rowstron. Topologyaware routing in structured peer-to-peer overlay networks, 2002.

[11] Condor. http://www.cs.wisc.edu/condor/.

[12] Aaron E. Darling, Lucas Carey, and Wu chun Feng. The design, implementation, and evaluation of mpiblast.

[13] M. Dayhoff, R.M. Schwartz, and B.C. Orcutt. *Atlas of protein sequence and structure*, volume 5, chapter "A model of evolutionary change in proteins.", pages 345–352. National Biomedical Research Foundation, Silver Spring, Maryland, 1978.

[14] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. pages 137–150, 2004.

[15] NCBI: National Center for Biotechnology Information. http://www.ncbi.nlm.nih.gov/.

[16] Bryan Ford, Dan Kegel, and Pyda Srisuresh. Peer-to-peer communication across network address translators. In *Proceedings of the 2005 USENIX Technical Conference*, 2005.

[17] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[18] Mark K. Gardner, Wu chun Feng, Jeremy Archuleta, Heshan Lin, and Xiaosong Ma. Grid applications—parallel genomic sequence-searching on an ad-hoc grid: experiences, lessons learned, and implications. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 104, New York, NY, USA, 2006. ACM Press.

[19] W. Gish. Wu-blast. http://blast.wustl.edu.

[20] Globus. http://www.globus.org.

[21] O. Gotoh. An improved algorithm for matching biological sequences. *J Mol Biol*, 162(3):705–708, December 1982.

[22] S. Henikoff and J. G. Henikoff. Amino acid substitution matrices from protein blocks. *Proc. Natl. Acad. Sci. U.S.A.*, 89(22):10915–10919, November 1992.

[23] Martin C. Herbordt, Josh Model, Yongfeng Gu, Bharat Sukhwani, and Tom VanCourt. Single pass, blast-like, approximate string matching on fpgas. In *FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*, pages 217–226, Washington, DC, USA, 2006. IEEE Computer Society.

[24] Tamer Kahveci and Ambuj K. Singh. Efficient index structures for string databases. In *The VLDB Journal*, pages 351–360, 2001.

[25] S. Karlin and S. F. Altschul. Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes. *Proc Natl Acad Sci U S A*, 87(6):2264–2268, March 1990.

[26] Karlin, Samuel, Dembo, Amir, and Kawabata, Tsutomu. Statistical composition of high-scoring segments from molecular sequences. *The Annals of Statistics*, 18(2):571–581, Jun 1990.

[27] Nicholas T. Karonis, Brian Toonen, and Ian Foster. Mpich-g2: a grid-enabled implementation of the message passing interface. *J. Parallel Distrib. Comput.*, 63(5):551–563, 2003.

[28] W. J. Kent. Blat–the blast-like alignment tool. *Genome Res*, 12(4):656–664, April 2002.

[29] Hong-Soog Kim, Hae-Jin Kim, and Dong-Soo Han. Hyper-blast: A parallelized blast on cluster system. In *International Conference on Computational Science*, pages 213–222, 2003.

[30] Derek Kisman, Ming Li, Bin Ma, and Li Wang. tpatternhunter: gapped, fast and sensitive translated homology search. *Bioinformatics*, 21(4):542–544, 2005.

[31] Arun Krishnan. Gridblast: a globus-based high-throughput implementation of blast in a grid computing framework: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(13):1607–1623, 2005.

[32] M. Li, B. Ma, D. Kisman, and J. Tromp. Patternhunter ii: highly sensitive and fast homology search. *Genome Inform. Ser. Workshop Genome Inform.*, 14:164–175, 2003.

[33] Heshan Lin, Xiaosong Ma, Praveen Chandramohan, Al Geist, and Nagiza F. Samatova. Efficient data access for parallel blast. In *IPDPS*, 2005.

[34] D. J. Lipman and W. R. Pearson. Rapid and sensitive protein similarity searches. *Science*, 227(4693):1435–1441, March 1985.

[35] B. Ma, J. Tromp, and M. Li. Patternhunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, March 2002.

[36] M. P. I. Message. Mpi: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[37] mpiBLAST G2. http://www.twgrid.org/global_resource/Project/mpi/index_html/.

[38] K. Muriki, K. D. Underwood, and R. Sass. Rc-blast: towards a portable, cost-effective open source hardware implementation. page 8, 2005.

[39] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, 48(3):443–453, March 1970.

[40] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proc Natl Acad Sci U S A*, 85(8):2444–2448, April 1988.

[41] Kevin T. Pedretti, Thomas L. Casavant, R. C. Braun, Todd E. Scheetz, C. L. Birkett, and Chad A. Roberts. Three complementary approaches to parallelization of local blast service on workstation clusters (invited paper). In *PaCT '999: Proceedings of the 5th International Conference on Parallel Computing Technologies*, pages 271–282, London, UK, 1999. Springer-Verlag.

[42] PowerBLAST. http://www.aethia.com.

[43] Human Genome Project. http://www.ornl.gov/sci/techresources/Human_Genome/home.shtml.

[44] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–??, 2001.

[45] Koichi Sasada. Yarv: yet another rubyvm: innovating the ruby interpreter. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 158–159, New York, NY, USA, 2005. ACM Press.

[46] R. Singh, W. Dettlo, V. Chi, D. Ho, m Tell, C. White, S. Altschul, and B. Erickson. Bioscan: A dynamically reconfigurable systolic array for biosequence analysis.

[47] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *J. Mol. Biol.*, 147(1):195–197, March 1981.

[48] Marcelo S. Sousa and Alba Cristina M. A. Melo. Packageblast: an adaptive multi-policy grid service for biological sequence comparison. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 156–160, New York, NY, USA, 2006. ACM Press.

[49] YAML Specification. http://yaml.org/spec/.

[50] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[51] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing mpi-io portably and with high performance. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 23–32, 1999.

[52] C. Thomas White, Raj K. Singh, Peter B. Reintjes, Jordan Lampe, Bruce W. Erickson, Wayne D. Dettloff, Vernon L. Chi, and Stephen F. Altschul. Bioscan: A vlsi-based system for biosequence analysis. In *ICCD '91: Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors*, pages 504–509, Washington, DC, USA, 1991. IEEE Computer Society.

[53] W. J. Wilbur and D. J. Lipman. Rapid similarity searches of nucleic acid and protein data banks. *Proc. Natl. Acad. Sci. U.S.A.*, 80:726–730, 1983.

[54] SWIG: Simplified Wrapper and Interface Generator. http://www.swig.org.

[55] J. Zhang and T. L. Madden. Powerblast: A new network blast application for interactive or automated sequence analysis and annotation. In *Genome Research 7*, pages 649–656, 1997.