ABSTRACT

LIN, HESHAN High Performance Parallel and Distributed Genomic Sequence Search. (Under the direction of Dr. Xiaosong Ma).

Genomic sequence database search identifies similarities between given query sequences and known sequences in a database. It forms a critical class of applications used widely and routinely in computational biology. Due to their wide application in diverse task settings, sequence search tools today are run on several types of parallel systems, including batch jobs on one or more supercomputers and interactive queries through web-based services. Despite successful parallelization of popular sequence search tools such as BLAST, in the past two decades the growth of sequence databases has outpaced that of computing hardware elements, making scalable and efficient parallel sequence search processing crucial in helping life scientists' dealing with the ever-increasing amount of genomic information.

In this thesis, we investigate efficient and scalable parallel and distributed sequence-search solutions by addressing unique problems and challenges in the aforementioned execution settings. Specifically, this thesis research 1) introduces parallel I/O techniques into sequence-search tools and proposes novel computation and I/O co-scheduling algorithms that enable genomic sequence search to scale efficiently on massively parallel computers; 2) presents a semantic based distributed I/O framework that leverages the application specific meta information to drastically reduce the amount of data transfer and thus enables distributed sequence searching collaboration in the global scale; 3) proposes a novel request scheduling technique for clustered sequence-search web servers that comprehensively takes into account both data locality and parallel search efficiency to optimize query response time under various server load levels and access scenarios. The efficacy of our proposed solutions has been verified on a broad range of parallel and distributed systems, including Peta-scale supercomputers, the NSF TeraGrid system, and small- or medium-sized clusters. In addition, our optimizations of massively parallel sequence search have been transformed into the official release of mpiBLAST-PIO, currently the only supported branch of mpiBLAST, a popular open-source sequence-search tool. mpiBLAST-PIO is able to achieve 93% parallel efficiency across 32,768 cores on the IBM Blue Gene/P supercomputer.

High Performance Parallel and Distributed Genomic Sequence Search

by
Heshan Lin

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fullfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2009

APPROVED BY:

Steffen Heber	Frank Mueller
Douglas Reeves	Nagiza Samatova
Xiaosong Ma Chair of Advisory Committee	

DEDICATION

To my parents *Guiwen Lin, Guiping Deng*
and
my wife *Xiaomin Lu*

BIOGRAPHY

Heshan Lin was born in Wuchuan, Guangdong, a small town located at almost the southernmost of China mainland in 1977. After his high school years, he moved to a lovely city named Guangzhou, an economic and cultural hub in Southern China, where he received his Bachelor of Science (B.S.) undergraduate degree in Applied Math from South China University of Technology. He then became a software engineer at a computer software startup working on distributed banking systems. Heshan joined the graduate program in the Department of Computer Science at Temple University in Fall 2001 and obtained his Masters degree in Computer Science in Spring 2004. He began the Ph.D. program in the department of Computer Science at North Carolina State University in Fall 2004. His research foucused on high-performance bioinformatics, parallel I/O, and distributed computing on desktop grids.

ACKNOWLEDGMENTS

This dissertation would not have been accomplished without the support and inspiration of many people.

First and foremost, I would like to express my sincere thanks and appreciation to my advisor, Dr. Xiaosong Ma, for her insightful guidance, consistent support, and patience in shaping me as a researcher from every perspective. She is always available for discussion even during her busiest schedule. I learned a lot from her on doing research as well as writing and presentation. I am much indebted to Dr. Nagiza Samatova for offering me a summer internship at Oak Ridge National Laboratory as well as mentoring me in the pioBLAST project during the early stage of my Ph.D. study. I would also like to express my gratitude to Dr. Wuchun Feng, who offered me three summer internships and invited me to join the mpiBLAST core development team. I want to thank Dr. Feng for his instrumental advices on the mpiBLAST-PIO project as well as many advices on how to improve my spoken and written communication skills.

I am grateful to Dr. Steffen Heber, Dr. Frank Mueller and Dr. Douglas Reeves for being a part of my dissertation committee. Their comments and questions helped me focus on important issues and avoid pitfalls during my research.

My deep appreciation also goes to many collaborators through my Ph.D. pursuit: Dr. Carlos Sosa, Dr. Pavan Balaji, Dr. Avery Ching, Dr. Mark Gardner and Jeremy Archuleta. I have benefited tremendously from our collaborations and the friendship we have kept since.

I would like to thank my great lab mates during the years: Jiangtian Li, Zhe Zhang, Alex Balik, Chao Wang, Tao Yang, Feng Ji, Amit Kulkarni, Prakash Ramaswamy, Nandan Tamineedi, Divya Dinakar and Sibin Mohan. I learned a lot from them through many spontaneous discussions and brainstorming sessions. Equally importantly, we had so much fun outside the work together which kept me refreshed from time to time.

I am grateful to Dr. John Blondin for his generosity in letting me use the Orbitty cluster. I want to thank Oak Ridge National Laboratory, National Energy Research Scientific Computing Center, Virginia Tech Advanced Research Computing, Ohio Supercomputing Center, High Performance Computing Center at North Carolina State University and IBM Rochester for granting me the access to their supercomputing facilities.

I sincerely acknowledge the funding resources of this dissertation work: 1) DOE ECPI Award (DE-FG02-05ER25685); 2) NSF CAREER Award (CNS-0546301); 3) Dr. Xiaosong Ma's joint appointment between North Carolina State University and Oak Ridge National Laboratory; 4) US DOE "Genomes to Life program" under the ORNL-PNNL project, "Exploratory Data Intensive Computing for Complex Biological Systems". 5) Los Alamos National Laboratory contract W-7405-ENG-36.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

In the past decades, research in bioinformatics has been dramatically accelerated by the ever-increasing compute power that helps people understand the composition and functional capability of biological entities and processes. A well-known outcome of the fusion between high-performance computing and high-throughput experimental biology was the assembly of the human genome by Celera Genomics using a cluster with nearly a thousand processors [1]. Computation-enabled breakthroughs like this have a tremendous impact on solving important problems in areas such as medical and environmental science.

One fundamental means to decipher the genomic information is through *sequence database searches*. A sequence database-search tool compares a set of query sequences against a database of DNA or amino-acid sequences using an alignment algorithm, and reports the statistically significant matches between the query sequences and the database sequences. The found similarities between a new sequence and a sequence of known functions can help identify the functions of the new sequence and find sibling species from a common ancestor. For instance, in 2003, sequence matching helped biologists identify the similarities between the recent SARS virus and the more well-studied coronaviruses, thus enhancing the biologists' ability to combat the new virus [2].

Thanks to the advent in the genome sequencing technology and the Internet for data collection/sharing, the amount of collective genomic sequence data has been doubling every 10 to 12 months [3, 4]. As shown in Figure 1.1, the size of GenBank [5], a widely used DNA sequence database maintained by the National Center for Biotechnology Information (NCBI), grew by over 5 orders of magnitude during the past two decades. Moreover,

it is evidential that the sequence acquisition will pace even faster in the near future [6]. Efficiently searching the ever-increasing sheer volume of sequence data has been a crucial task to many computational biology studies. Due to their wide application in diverse task settings, sequence search tools today are run on several types of high-performance computing environments, including batched jobs on one supercomputer or across multiple geographically distributed supercomputers as well as interactive jobs handled through clustered web services. In this thesis, we investigate efficient and scalable parallel and distributed sequence-search solutions to address the unique problems and challenges in those various execution settings.

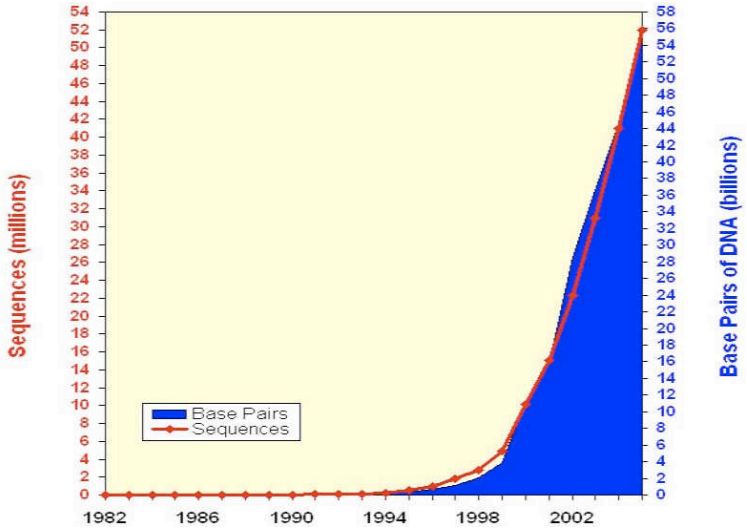


Figure 1.1: GenBank growth

1.1 Scaling Genomic Sequence Search on Massively Parallel Computers

With the explosive growth of sequence data, genomic sequence search has become one of the most compute- and data-intensive applications in scientific computing. While the next generation of massively parallel computers can provide the necessary horsepower for tackling the long running sequence-search jobs, existing parallel sequence-search tools

are designed for processing moderate-sized databases on small to medium sized clusters, and hence are not ready to fully take advantages of modern supercomputers.

Specifically, most existing parallel sequence-search tools mainly focus on parallelizing the alignment computation, with the data management issue largely overlooked. Our study revealed that the data management tasks such as merging and writing the output data can significantly limit the program's scalability in existing sequence-search tools. In addition, unlike numerical simulations that are commonly seen in traditional scientific computing, genomic sequence search possesses irregular computation and I/O patterns. The computation cost of each subtask is hard to predict, and the output data generated at each process is non-contiguous with non-uniform size distributions. These runtime irregularities are not properly addressed in existing parallel sequence-search tools, which can lead to serious resource under-utilization on large-scale deployments.

In this work, we first introduce parallel I/O to remove the data management bottleneck in existing parallel sequence-search tools when processing large databases. We then address the runtime irregularities of sequence-search applications with an integrated scheduling algorithm that gracefully coordinates dynamic computation load-balancing and asynchronous high-throughput parallel I/O. The proposed scheduling approach greatly improves the scalability of massively parallel genomic sequence searching.

1.2 Semantics-based Distributed I/O with the ParaMEDIC Framework

The compute and storage requirements of large sequence-search jobs in advanced computational biology are greater than ever before. Most life scientists do not have the local access to the supercomputing resources needed for their search jobs. Thus the sequence-search results generated on the remote supercomputers need to be frequently offloaded to local for visualization and further analysis. Moreover, highly demanding sequence-search jobs such as database-to-database alignment may require compute resources from tens of thousands of processors and generate hundreds of terabytes of data. For these jobs, few supercomputers will be simultaneously equipped with the necessary compute and storage resources. Efficient data movement between distributed computation resources connected with wide area networks is critical to search results offloading and large-scale collaborative

efforts of sequence analysis.

There has a lot of investments in high-speed distributed network connectivity to alleviate issues related to moving massive data across supercomputing sites. However, these high-speed networks do not provide an end-to-end connectivity to a very high percentage of scientific community. In addition, the amount of data generated by large sequence-search jobs is so large that even at 100% network efficiency, the I/O time can dominate the overall execution time.

In this work, we present a framework called "ParaMEDIC: Parallel Metadata Environment for Distributed I/O and Computing" which uses sequence-search semantic information to convert the generated data to orders-of-magnitude smaller metadata at the compute site, transfers the metadata to the storage site, and re-processes the metadata at the storage site to regenerate the output. ParaMEDIC can drastically reduce the amount of data that need to be transfered over the network, making global-scale distributed sequence search feasible.

1.3 Adaptive Request Scheduling of Parallel Sequence-Search Web Servers

Besides large, resource-demanding sequence-search jobs, life scientists routinely need to search a small number of query sequences against public sequence databases in the sequence analysis work flow. For those small search jobs, the batch parallel processing model does not necessary provide the most convenient interface and the desired quick response time to end users. Many scientists would prefer to use the so-called service-oriented infrastructure [7], where data analysis is provided through the web interface and carried out on dedicated parallel computing resources (*e.g.* clusters) behind the web server. As this approach can hide the complexity of parallel job management from biologists as well as enable the effective utilizing and sharing of high-end computation resources, many research institutions are moving to hosting online genomic sequence analysis.

Unlike with general-purpose clustered web servers, where servicing a request consumes only a small amount of computation and can be efficiently accomplished on a single processor, with sequence search servers, servicing a query requires substantial parallel processing and data access. Therefore, existing scheduling techniques used in content-serving

clustered web servers cannot be directly applied to this new context. Meanwhile, although space-sharing job scheduling on distributed memory machines has been well studied, existing algorithms do not address the sharing of storage resources between jobs. Further, there lacks a comprehensive scheduling scheme that coordinates the decomposition and mapping of computation tasks, and those of the data accesses.

In this work, we propose an adaptive request scheduling algorithm that extends and integrates several existing algorithms from the content-serving web scheduling and the space-sharing parallel job scheduling. The novel combination of these algorithms allows a sequence-search web server to automatically react to the variation of system load and query access pattern for optimized query response time by comprehensively taking into account data locality and parallel efficiency in making scheduling decisions.

1.4 Contributions

We consider the major contributions of this thesis research as:

- Proposing and designing an end-to-end efficient data management framework that highly improves the scalability of searching against large sequence databases [8, 9].
- Proposing, designing and evaluating scalable computation and I/O scheduling solutions for massively parallel sequence search on a variety of state-of-the-art supercomputers [10, 11].
- Developing and delivering the mpiBLAST-PIO software, an official release branch of mpiBLAST, which is a widely used open-source sequence-search tool. mpiBLAST-PIO was able to achieve 93% parallel efficiency across 32,768 cores on the IBM Blue Gene/P supercomputer.
- Collaborating with researchers from Argonne National Laboratory and Virginia Tech to build ParaMEDIC, a semantic-based distributed I/O framework that enables efficient large-scale sequence search over globally distributed supercomputing resources [12]. The ParaMEDIC project resulted in a HPC Storage Challenge Award at SC07 and Distinguished Paper Award in ISC08 [13].

- Proposing, designing, and evaluating an adaptive query scheduling algorithm for clustered sequence-search web servers. The proposed algorithm can automatically adapt system configurations for optimized query response time under various system loads and query patterns [14, 15].

While our thesis research focuses on genomic sequence-search applications, we believe that many solutions and design rationales from this research can be applied to a broad class of data-intensive scientific applications.

1.5 Organization

The rest of this dissertation is organized as follows. Chapter 2 describes the background information of genomic sequence database search. Chapter 3 discusses the I/O and scheduling optimizations for massively parallel genomic sequence search. The ParaMEDIC distributed I/O framework is discussed in Chapter 4. Chapter 5 presents adaptive request scheduling algorithms for high-performance online sequence-search servers. Chapter 6 surveys the related work. Finally, Chapter 7 concludes the thesis.

Chapter 2

Sequence Database Search Background

2.1 Sequence Databases

Unlike relational databases used widely in commercial systems, most biological databases are flat files containing strings of nucleotides (guanine, adenine, thymine, cytosine, and uracil) and/or amino acids (threonine, serine, glycine, etc.) Each sequence of nucleotides or amino acids represents a specific gene or protein (or a section thereof) respectively. Although sequences have complex, dynamic and multi-dimensional structure, they are represented in shorthand, using single letters to denote nucleotide or amino acid residues. For example, a DNA sequence is encoded as a string of characters A, C, G or T. This abstraction makes it convenient for storing, publishing and sharing the sequence data.

There are many public sequence databases that are widely used by computational biology researchers. GenBank [16] is an annotated collection of publicly available DNA sequences maintained by the National Center for Biotechnology Information (NCBI) [17]. It is the union of the DNA Data Bank of Japan (DDBJ) [18] and the European Molecular Biology Laboratory (EMBL) [19] nucleotide database from the European Bioinformatics Institute (EMI) [20]. These three organizations exchange data every day to keep their local copies synchronized. SWISS-PROT [21] is a curated protein sequence database jointly managed by the Swiss Institute of Bioinformatics (SIB) [22] and the European Bioinformatics Institute (EBI) [23]. Protein Data Bank (PDB) [24] is a well-established structure database

containing structures of proteins, nucleic acids and a few carbohydrates.

2.2 Sequence Database Search Tools

Sequence database search is one of the fundamental tasks routinely performed in many computational biology research areas. A typical use of sequence search is to find similarities between newly discovered sequences and those in known nucleotide or protein databases. The searched results can then be used to predict the structures and functions of new sequences. They also allow people to estimate the evolution distance in phylogeny reconstruction and perform gnome alignments.

A sequence database-search tool compares a set of query sequences against a database of DNA or amino-acid sequences using an alignment algorithm, and reports the statistically significant matches between the query sequences and the database sequences. Sequence alignment algorithms have been extensively studied during the past decades. The Needleman-Wunsch algorithm [25] finds optimal global alignments of two sequences by maximizing the number of matched residues and minimizing the gaps necessary in aligning two sequences. Smith-Waterman [26] solves the local alignment problem with a dynamic programming algorithm. It searches every possible match position between two sequences and is highly sensitive in comparison quality. However, it is too computationally expensive to be directly applied to large databases searches. FASTA [27] and the BLAST family of alignment programs, including NCBI BLAST [28, 29], MegaBLAST [30] and WU-BLAST [28, 31, 32], speed up the alignment computation with seed-and-extend heuristic methods and make searching large sequence databases more practical. Despite different trade-offs between the search accuracy and efficiency, these seed-and-extend algorithms are similar and have quadratic computational complexity.

Currently, the BLAST family of algorithms are the de facto standard tools. In fact, it is estimated that 75% to 90% CPU cycles used in life sciences are spent on this family of applications [33]. However, they have difficulty in keeping up with the growing rate of sequence acquisition [34]. This problem is addressed in two orthogonal directions: (1) to develop faster and smarter sequence alignment algorithms (e.g., SSAHA [35], PatternHunter [36], and BLAT [34]), and (2) to take advantage of parallel processing. Our research mainly focuses on efficient parallel and distributed computing solutions for general sequence-search

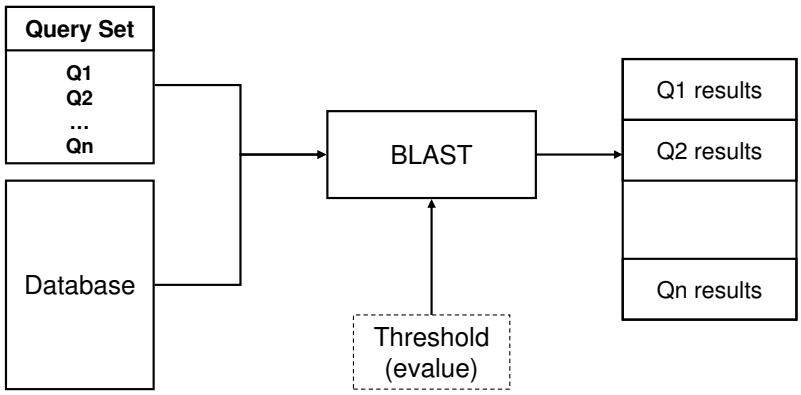


Figure 2.1: BLAST Search

tools.

Our research will use the popular BLAST search tool as a studay case. Figure 2.1 shows the diagram of a typical BLAST search. The input includes a *query set* supplied by the user, which is a collection of nucleotide or protein sequences to be searched, and a sequence database to be searched against. Figure 2.2 gives a sample section of BLAST input. In both the query set and the database, the first line of a sequence starts with a right arrow symbol indicating the sequence header, which includes information such as the sequence ID and the sequence description etc. BLAST performs Cartesian-product-style comparison between query sequences and database sequences. The results are printed to the standard output or an output file, organized by the query sequences in the order they were given in the query set. For each query sequence, the result data contains *alignments* between this sequence and database sequences that have closeness beyond a given threshold. The closeness is measured by *evalue* which is computed by the BLAST algorithm to represent the degree of similarities between two sequence segments. A partial sample output of searching query sequence "Query1" against the database in Figure 2.2 is given in Figure 2.3. There are two sections in the output: a one-line summary and the alignments information. In the one-line summary section, a set of result database sequences are reported to be similar to the query sequence, ordered by their evalues. The actual alignments (segments of sequences where similarity is identified) between the query sequence and result database sequences are presented in the alignments section.

Although sequence search tools use different algorithms and optimization tech-

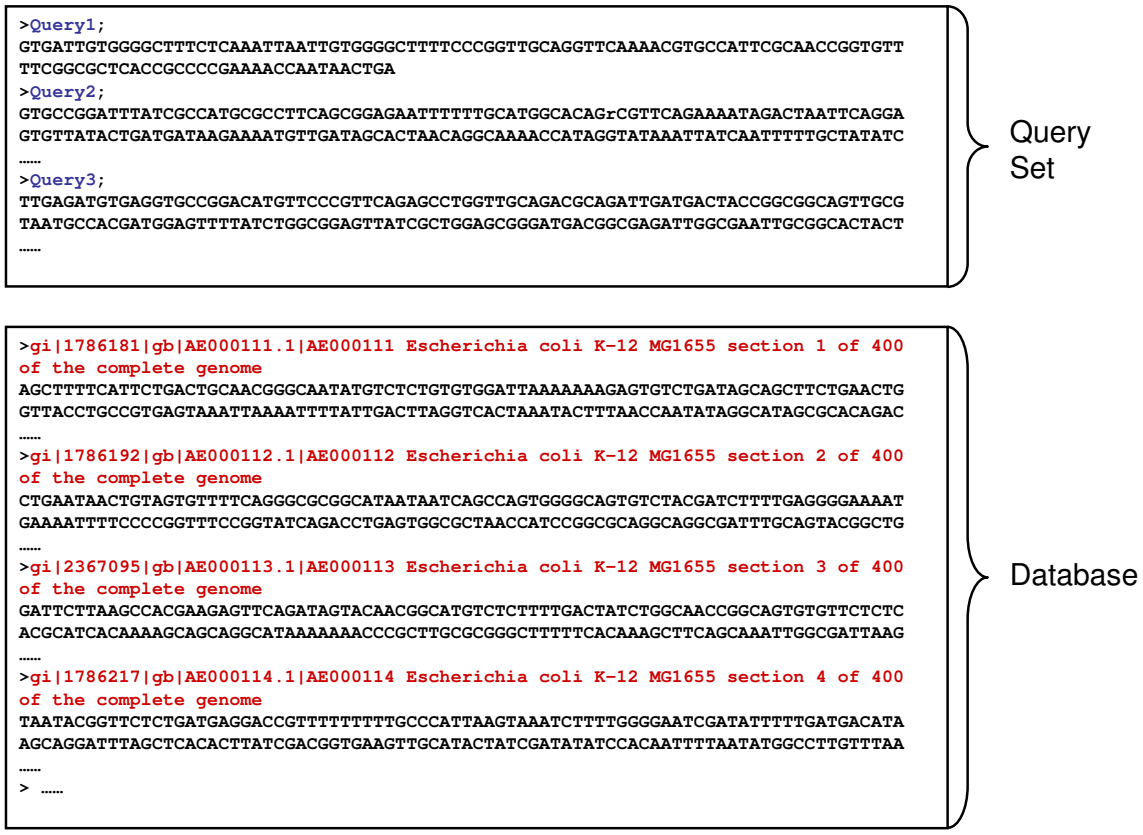


Figure 2.2: BLAST Input

niques, they have certain common characteristics. First, the input of these tools consists of a set of query sequences and a sequence database. Second, they perform pairwise comparison between the query set and the database. Finally, the output of each query sequence contains similar database sequence segments ordered by the degree of similarity. These common characteristics enable the development of a general-purpose optimization framework for this type of applications.

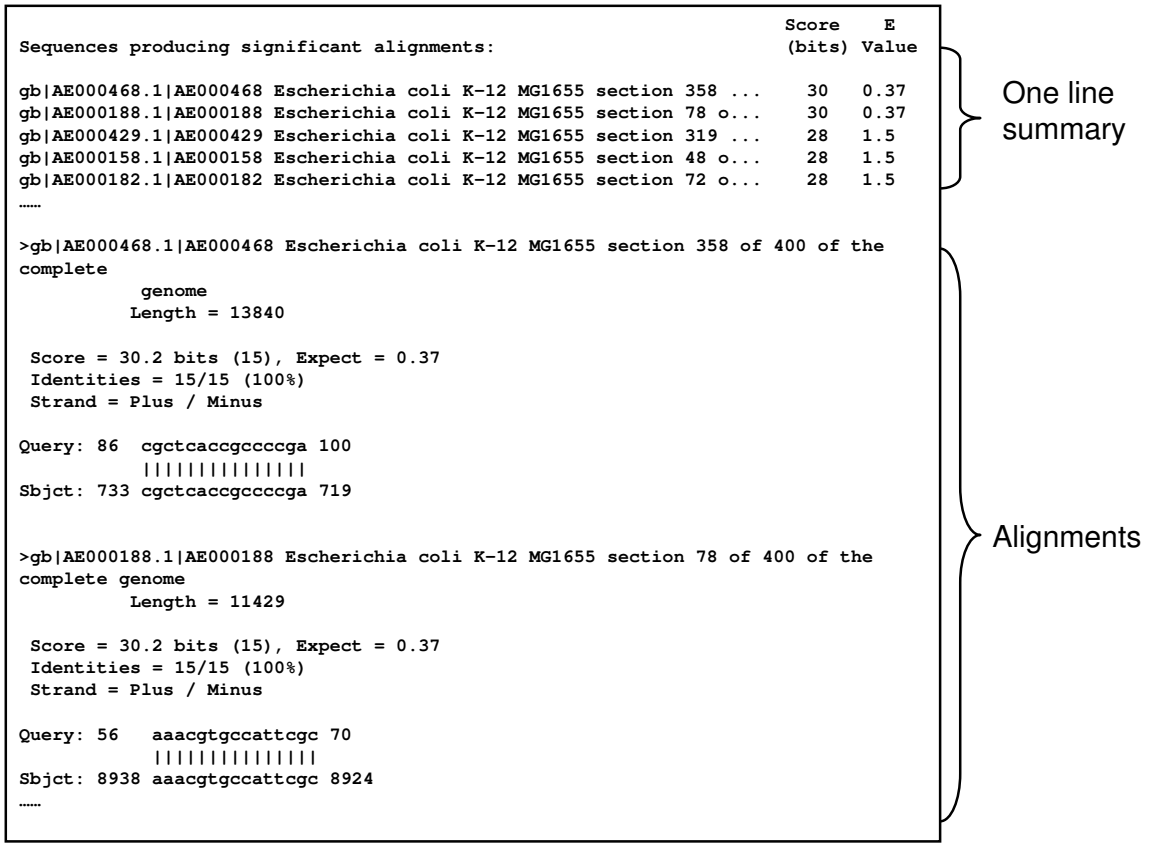


Figure 2.3: BLAST Output

Chapter 3

Co-scheduling Computation and I/O for Massively Parallel Genomic Sequence Search

3.1 Introduction

Today, the collective amount of genomic information is now doubling every 10-12 months [4, 3], while the computational horsepower of a single processor is only doubling every 18-24 months. The widening disparity between the computational demand of analyzing the collective genomic data and the uniprocessor's processing power keeps challenging the scalability and efficiency of parallel sequence-search tools. Initial studies have shown that it takes hours to days on thousands of processors to complete large sequence-search jobs such as genome-to-genome comparisons [37] and database-to-database alignments [38]. With Moore's Law now switching to doubling the number of cores per chip instead of doubling the uniprocessor performance in every silicon generation [39], it will soon become critical for the life scientists to leverage massively parallel computers to solve their time-consuming sequence-matching problems. It is thus imperative for sequence-search applications to be able to scale efficiently on the state-of-the-art supercomputers.

Our past experiences suggested that parallel sequence-search possesses high irregularities in both computation and I/O patterns:

- The execution time of a search task is hard to predict from the simple metrics such as the size of the input data. Tasks processing a same amount of input can have execution time differing by *orders of magnitude* [38].

- The output data distribution on different processes is fine-grained as well as irregular, and varies from one query to another depending on the search result [9].

The reason for these runtime irregularities is twofold. First, the amount of required compute resources and the output data distribution of a search task are dependent on the similarities between the compared sequences, which can vary significantly even between different tasks processing a same amount of input data. Second, popular sequence alignment algorithms such as BLAST [28, 29] employ heuristics to improve computational efficiency, making it hard to predict the execution time of a search task. How to efficiently handle the runtime irregularities is a major challenge in designing scalable sequence-search applications on massively parallel computers.

Efficient scheduling for irregular scientific applications has been extensively investigated since the last decade, with a wealth of techniques proposed for dynamic load-balancing by leveraging applications' runtime profiles [40, 41, 42, 43] or probabilistic signatures [44, 45, 46, 47, 48]. Most of these scheduling studies, however, focused on compute-intensive applications and did not address the irregular I/O issue of data-intensive applications. Existing studies on noncontiguous I/O optimizations [49, 50, 51, 52, 53, 54], on the other hand, emphasized on improving programs' actual I/O performance and were often evaluated without considering the computation scheduling. Thus, the interaction between the I/O optimizations and the computation scheduling, especially for irregular scientific applications, has not been well understood.

In this work, we systematically investigate the computation and I/O scheduling for genomic sequence-search applications. We consider our contributions as follows:

- Our study revealed that for data-intensive applications with irregular computational kernels such as genomic sequence search, the incoordination between the I/O optimization and the computation scheduling can result in severe performance degradations. Consequently, we proposed an integrated scheduling approach that gracefully coordinates fine-grained, dynamic computation load-balancing and asynchronous output processing to maximize the sequence-search throughput.

- We proposed a portable, asynchronous two-phase I/O technique for writing noncontiguous data in irregular, data-intensive applications. This I/O approach can obtain the performance benefit of parallel I/O without synchronization overhead imposed by traditional collective I/O techniques.

- We realized our scheduling approach on top of mpiBLAST [55], a popular open-source, parallel sequence-search tool, and developed a research prototype named mpiBLAST-PIO. The efficacy and portability of our scheduling optimizations were evaluated on three general-purpose parallel computers (configurations to be described in Section 3.4). MpiBLAST-PIO scaled perfectly to hundreds of processors on the three test parallel platforms. We also conducted a case study on the IBM Blue Gene/P massively parallel computer. The performance results shown MpiBLAST-PIO achieved 93% parallel efficiency across 32,768 cores on Blue Gene/P.

3.2 MpiBLAST Background

MpiBLAST [55] is an open-source sequence-search tool that parallelizes the NCBI BLAST toolkit [28]. The original design of mpiBLAST follows the *database segmentation* approach. Specifically, MpiBLAST organizes parallel processes into one master and many workers. The master uses a greedy algorithm to assign pre-partitioned database fragments to workers. The workers copy the assigned fragments to their local disks (if available) and perform BLAST search concurrently. Upon finishing searching one fragment, a worker reports its local results to the master for centralized result merging. The above process repeats until all the fragments have been completed. Once the master receives results from all the workers for a query sequence, it calls the standard NCBI BLAST output function to format and print out results to an output file. MpiBLAST achieves good speedup when the number of processes is small or moderate, by fitting the database into main memory and eliminating repeated scanning of disk-resident database files.

The scalability of the original mpiBLAST design is greatly hampered on massively parallel computers for two reasons. First, it exploits parallelism only through database partitioning. This will result in many tiny database fragments and involve significant parallelizing overhead on large-scale deployments. Second, the scheduling and results processing are performed on a single master, which can easily become a bottleneck as system size grows.

3.3 Integrated Computation and I/O Scheduling

In this section, we first present the software architecture of mpiBLAST-PIO. Then we discuss the details of our proposed computation and I/O scheduling optimizations.

3.3.1 Software Architecture

MpiBLAST-PIO adopts a hierarchical architecture as depicted in Figure 3.1. At the top level, the system is organized into equal-sized *partitions*, which are supervised by a dedicated *supermaster* process. The supermaster is responsible for assigning tasks to different partitions and handling inter-partition load balancing. Within each partition, there is one master process and many worker processes. The master is responsible for coordinating both computation and I/O scheduling in a partition. It periodically fetches a subset of query sequences (defined as a *query segment*) from supermaster and assigns them to workers, as well as coordinates the output processing of queries that have been processed in the partition. Such a hierarchical design avoids creating scheduling bottleneck as the system size grows by distributing the scheduling loads on multiple masters.

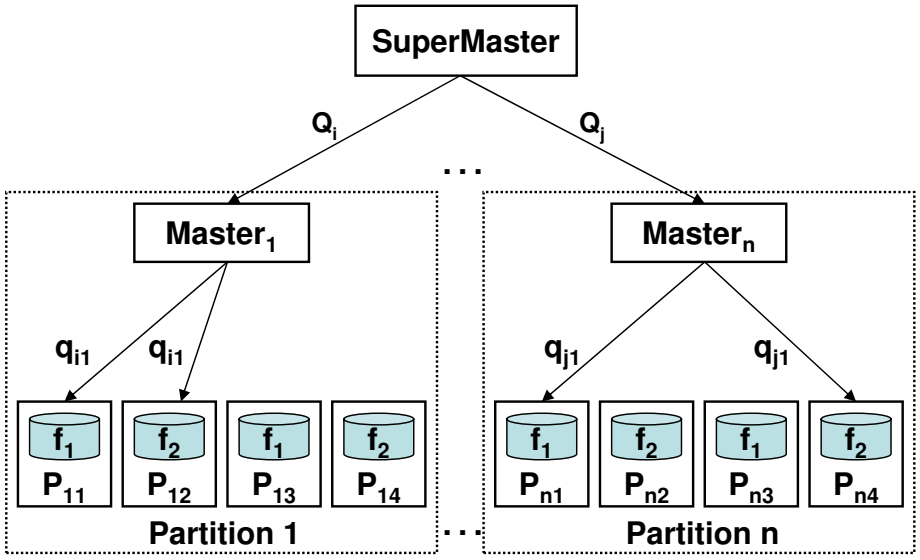


Figure 3.1: mpiBLAST-PIO software architecture. Q_i and Q_j are query segments fetched from the supermaster to masters, and q_{i1} and q_{j1} are query sequences that are assigned by masters to their workers.

In this architecture, the compute processes in the system are segregated into two

groups – masters and workers. In order to maximize the system throughput, it is important to keep both groups of processes equally busy so that the system idleness is minimized. The key to balance loads between masters and workers is to choose an appropriate partition size S_p (defined as the number of workers in the partition). To this end, our design supports mapping an arbitrary number of workers to a master and allows users to determine the appropriate S_p value through initial profiling with sampled sequences from the original query[1]. As the master loads will increase monopoly as S_p grows, a sweet point can be found by comparing the program performance at gradually increased S_p values. Note that the optimal S_p value is platform- and workload-dependent, and automatic tuning of the parameter is out of the scope of this work.

To enable database segmentation, the sequence databases are pre-partitioned into fragments and stored on the shared file system. MpiBLAST-PIO defines two running modes, *non-sharing* and *sharing*, which distribute the database fragments to the worker nodes differently.

The *non-sharing* mode assumes input database fragments are not shared between different parallel BLAST jobs, making it suitable for platforms without locally attached disks, such as IBM Blue Gene systems. In this mode, the fragments are replicated to worker nodes' memory in a way similar to that used in previous parallel BLAST studies [56, 10]. During the system initialization, all workers in the system are organized into temporary equal-sized replication groups, and the first group will be designated as the I/O group. All database fragments are assigned to the workers in the I/O group in a round-robin fashion. Each worker in the I/O group then reads in its assigned fragments in parallel and broadcasts them to the corresponding workers in all other groups.

In institutions where BLAST is heavily used by many users and the cluster nodes are equipped with locally attached disks, it is desirable to enable sharing of common sequence databases between BLAST jobs to reduce the cost of data movements. In the *sharing mode*, the database fragments used by a BLAST job will remain on the local disks of worker nodes after the job is finished. At the beginning of execution, workers report the cached fragments on their local disks to the master. These fragments distribution information will be taken in to account in the scheduling decision.

[1]We found that using randomly sampled query sequences from a BLAST job to perform initial profiling is practical in finding appropriate parameter values in our system.

3.3.2 Fine-grained, Dynamically Load-balanced Computation Scheduling

BLAST search time is highly variable and unpredictable as found in our past research [38]. To our best knowledge, there is no effective way to estimate the execution time of a given BLAST search in the existing literature. Without a priori knowledge of queries' processing time, using a greedy scheduling algorithm to assign fine-grained tasks to idle processes seems to be sufficient to load balancing.

To effectively achieve load-balance across multiple partitions, assigning a small query segment to a partition at a time is desirable, especially for running on supercomputers with a large number of partitions. On the other hand, fine-grained query-segment allocation incurs two problems. First, the scheduling overhead increases with smaller query segment sizes. Second, using small query segment forces frequent synchronization between the workers within each partition, leaving faster workers waiting for its slowest peer to finish before acquiring a new segment of queries to work on. In particular, with small query segments, there are not enough queries to "cancel out" the per-query imbalance of search time, therefore the intra-partition load-imbalance worsens and the partition-wise resource utilization degrades.

We address the above problems associated with small scheduling granularity by using *task-oriented scheduling* and *query prefetching* at each partition. With task-oriented scheduling, rather than assigning each query to a fixed group of workers, its search is broken into a set of tasks corresponding to the set of database fragments this query has to be searched against. The masters dynamically maintain a window of outstanding tasks. Whenever a worker is done with its current task, it contacts the master to request another one. The set of workers that work on one particular query is thus dynamically formed. With query prefetching, the master requests the next query segment when the total number of outstanding tasks falls under a certain threshold. By combining these two techniques, workers will not be slowed down by waiting for its peers or for the next batch of query sequences.

The scheduling process running on the master is given in Algorithm 1. The master maintains a list of query sequences, QL, that are being processed in the partition. A query sequence in QL is corresponding to $|F|$ individual tasks, each searching the query sequence against a distinct database fragment. The master keeps tracking how many tasks have been

completed by workers. When observing that the number of total uncompleted tasks of all query sequences in QL is less than the number of workers ($|W|$) in the partition, the master issues a query prefetching request to the supermaster. To overlap network communications with local job scheduling, the master receives the query segment in the background with a nonblocking MPI call. The new query segment received from the supermaster will be appended to the end of QL.

In the above design, the size of a prefetched query segment (S_q) is configurable. Using smaller S_q can yield better load balance results but increase the number of messages sent to the supermaster. In practice, S_q can be configured to an arbitrarily small value as long as the supermaster is not overloaded by the prefetcting messages. According to our experiments on the System X cluster (configuration details will be described in Section 3.4) using 1024 processors, for typical BLAST searches the supermaster is not a performance bottleneck even when the S_q is set to 1. In our case study that specially optimized for the IBM Blue Gene/P system (to be presented in Section 3.5), we found setting S_q to 5 was sufficient in a scale of 32,768 processes.

At the inner-partition level, workers periodically report to the master for assignments when idle. Upon receiving a task request from an idle worker (w_j), the master scans QL in the FIFO order to determine a task for w_j as follows. For the current query sequence being examined (q_c):

1. If w_j has cached some database fragments that have not been searched against q_c, the cached fragment that is least distributed (i.e., cached by fewest workers) in the partition will be assigned to the worker.

2. If w_j has not cached any unsearched fragment of query sequence c and the sharing mode is used, the least-distributed fragment is assigned to w_j, who will then load the assigned fragment into its local cache before the search.

Here data locality is taken into account to reduce data movements and keep partition-wide data distribution to a minimum under the sharing mode. If no tasks can be found for this worker, the scheduling algorithm move on to the next query in QL. The same scheduling procedures are repeated until a task is decided for the idle worker or until all unfinished query sequences in QL have been examined, in which case the worker has finished its own portion of work. By allowing uncompleted tasks to be independently scheduled to any

Algorithm 1 Master Scheduling Algorithm

11.0

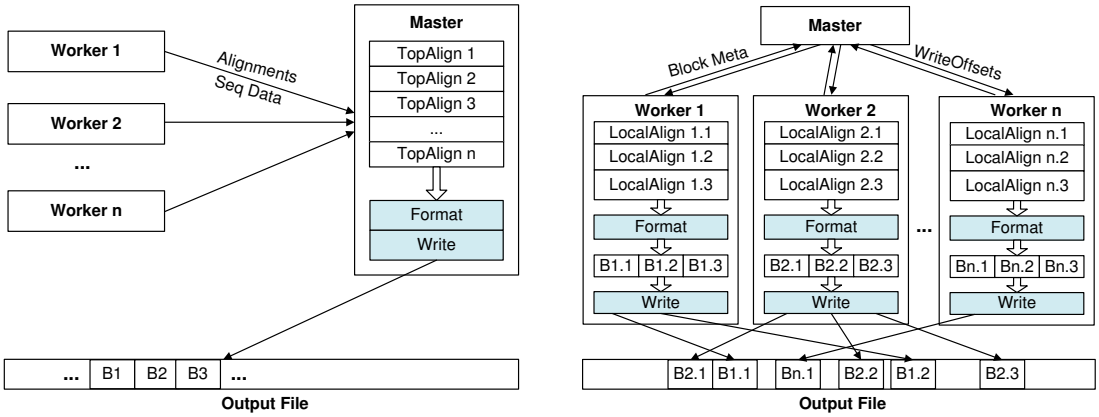
Let $QL = \{q_1, q_2, ...\}$ be the list of unfinished query sequences
Let $F = \{f_1, f_2, ...\}$ be the set of database fragments
Let $Unassigned_i \subseteq F$ be the set of unassigned database fragments for query sequence q_i
Let $W = \{w_1, w_2, ...\}$ be the set of workers in this partition
Let $D_i \subseteq W$ be the set of workers that cached fragment f_i
Let $Distributed = \{D_1, D_2, ...\}$ be the set of D for each database fragment
Let $C_i \subseteq F$ be the database fragments cached by worker w_i
Let $assignment_i$ refer to the assignment to the i^{th} worker

Require: $|W| \neq 0$
  while not all query sequences have been finished **do**
    if number of all unassigned fragments in $QL < |W|$ **then**
      Issue segment prefetching request to supermaster
    end if
    if Received a query segment QS from supermaster **then**
      for $q_i \in QS$ **do**
        Append q_i to QL
        $Unassigned_i \leftarrow F$
      end for
    end if
    Receive task request from worker w_j
    $q_c \leftarrow QL.head$
    $assignment_j \leftarrow \ < \varnothing, 0 >$
    while $q_c \neq QL.tail$ and $assignment_j = \ < \varnothing, 0 >$ **do**
      if $\exists f_i \in Unassigned_c$ and $w_j \in D_i$ **then**
        Find f_k such that $min_{D_k \subseteq Distributed}|D_k|$
          and $w_j \in D_k$ and $f_k \in Unassigned_c$
        $assignment_j \leftarrow \ < q_c, f_k >$
      else
        if sharing mode is used **then**
          Find f_k such that $min_{D_k \subseteq Distributed}|D_k|$
            and $f_k \in Unassigned_c$
          $assignment_j \leftarrow \ < q_c, f_k >$
        end if
      end if
      if $|Unassigned_c| = 0$ **then**
        $QL.head \leftarrow QL.head.next$
      end if
      $q_c \leftarrow q_c.next$
    end while
  end while

workers that have cached the corresponding database fragments, our scheduling algorithm helps balance loads between workers even the search times of different tasks are highly skewed.

3.3.3 Scalable Distributed Results Processing

While most existing studies on parallel sequence search have focused on the parallelization of the sequence alignment computation, we have found that efficient handling of output data is crucial in sustaining the parallel execution efficiency when scaling to a large number of processors. In this section, we identify several performance issues in mpiBLAST's original result-processing protocol. We then present a light-weighted result-processing protocol used in mpiBLAST-PIO. The new propotocol can significantly reduce the non-search overhead of sequence-search tools using the *database segmentation* approach as well as enables efficient processing of output-intensive queries.



(a) Centralized Results-Processing (b) Distributed Results-Processing

Figure 3.2: Compare centralized and distributed design of results processing. In the centralized design, the results formatting and writing is serially done by the master. While in the distributed design, the results formatting and writing is concurrently done by workers.

As discussed in Section 3.2, originally mpiBLAST adopts a centralized result-processing approach to merge results generated at different workers. Specifically, In mpiBLAST, a worker produces *result alignments* after searching a query sequence against a database fragment. Each of such result alignments is a piece of intermediate result describing a hit area identified from an in-database sequence. Information regarding an alignment,

such as sequence IDs, evalue and the locations of the hit is stored in a per-alignment data structure. Figure 3.2(a) depicts the procedures of centralized results processing. When a search task is finished, the worker sends the result alignments together with the corresponding sequence data to the master. The result alignments belonging to the same query will be merged into a list in the order of their evalues, and the corresponding sequence data will be buffered and used later in the results formatting. A query is ready for output when the result alignments of all database fragments have been received. The result data of multiple ready queries are processed and written in their submission order. To process a ready query, the master calls the output routine of NCBI BLAST, which in turn formats each qualified alignment (in the top k range of the alignment list) and appends the corresponding result data block to the output file.

The above centralized design is based on the assumption that the results formatting and writing can be easily handled by a single compute node. This assumption, however, is not valid given the ever-growing scale of sequence databases, query workload, and parallel computers. For example, researchers have found that searching individual "hard" queries against large DNA sequence databases could yield gigabytes of output data [38]. As a result, centralized results merging and formatting become the major scalability bottleneck in mpiBLAST. Figure 3.3 shows the execution breakdown of searching 300 `nr` sequences against the database itself with mpiBLAST v1.4 on System X at Virginia Tech (configurations to be described in Section 3.4). The "search time" refers to the average time spent on the actual BLAST search algorithm by each worker. The "other time" includes all parallel overhead, which is dominated by the results processing cost at the scale of our experiments. As shown in Figure 3.3, the search time decreases nicely as more workers are used, but the non-search overhead also increases rapidly. Consequently, the overall execution time stops decreasing at 32 workers.

Several reasons account for the poor scalability of centralized results processing. First, all result alignments need to be buffered at the master before output, imposing a high memory demand on this single node, causing serious performance degradation, or simply forbidding the completion of certain output-intensive queries. Second, the results formatting and writing are performed sequentially, making the master a potential performance bottleneck in handling a bulky result volume. Finally, the result sequence data need to be sent over the network to the master for preparing output, adding high message-passing cost

to the application-visible overhead.

Figure 3.3: Searching 300 randomly sampled `nr` sequences against `nr` itself with mpiBLAST-v1.4 on the System X cluster. The `nr` database is partitioned into 8 fragments, and a replica of the database is pre-distributed to every 4 workers in a round-robin fashion.

To address this problem, mpiBLAST-PIO adopts a distributed results-processing design to enhance its scalability in searching both individual and multiple queries on a large number of processors. Figure 3.2(b) illustrates this new output workflow. First, after generating local result alignments, workers take one more step to format the alignments into output blocks and store them in memory buffers. Each output block is stored with the evalue of the corresponding alignment. Next, workers submit *block metadata*, which consists of the evalue and size of each local output block, to the master. The master then merges and filters out output blocks of unqualified alignments according to their evalues. When the block metadata of all workers have been received for one query, the master calculates the in-file offsets of globally qualified output blocks and sends the those back to the workers who buffered corresponding output blocks. Now knowing the subset of their qualified local output blocks, the workers will write the output buffers that they already prepared out to the file system. These write operations can be carried out in parallel using several strategies, to be discussed in Section 3.3.4.

The above distributed results-processing greatly reduces the parallel overhead compared to the centralized scheme. First, it improves the level of parallelism by shifting the bulk of work in results formatting from the master to the workers and allows output preparation to proceed in parallel. Second, it alleviates the memory space bottleneck at the

master node by having all workers collaboratively buffer intermediate results. Third, only a small amount of data (i.e., evalues, sizes and write offsets of output blocks) needs to be exchanged between the master and the workers, dramatically reducing the communication volume in the system. Although the new scheme has the workers format all their intermediate results regardless of whether these results will be included in the global output, the wasted processing is outweighed by the benefit of saving transferring bulky sequence data and parallelizing the output preparation.

3.3.4 Parallel Output Scheduling

Our dynamic computation scheduling and distributed results processing leave each involved worker a set of non-contiguous output data blocks to write to disjoint ranges in the output file. How to efficiently write those output data to the file system is another challenge to sustaining high sequence-search throughput.

The optimizations of non-contiguous I/O operations have been well studied for parallel numerical simulations, which often possess predictable data access patterns and balanced computation models. However, in our situation the unique aspects of parallel sequence-search applications complicate the I/O design in several ways:

- The output data distribution is fine-grained as well as irregular, and varies from one query to another depending on the search result [38]. Straightforward, uncoordinated I/O can result in poor I/O performance.

- Unlike in timestep simulations, where the computation time is well balanced across processes, here the computation time could be significantly imbalanced across workers searching the same query on different database fragments. In addition, there is no inherent synchronization in the computation core between searching different queries. Synchronous parallel I/O techniques may incur high parallel overhead and have negative impacts on our load balancing algorithms.

The above observations suggest that traditional non-contiguous I/O optimization techniques, specifically data sieving and collective I/O (described in Section 6.2), may not be suitable for massively parallel sequence search. In this paper, we investigate an alternative I/O optimization which employs an asynchronous, two-phase writing technique. We compares it with existing parallel I/O optimizations by evaluating four alternative output

strategies: *WorkerIndividual*, *WorkerCollective*, *MasterMerge* and *WorkerMerge* (as illustrated side by side in Figure 3.4). Among them, the first three are based on existing I/O techniques, and the last one (WorkerMerge) is based on our proposed I/O optimization.

WorkerIndividual

As described in Section 5.4, once the workers receive write offsets of buffered output blocks from the master, they can go ahead and issue write requests to the shared file system to write out the buffered output blocks. Figure 3.4(a) depicts the procedure of the WorkerIndividual strategy with an example setting consisting of three workers, assuming the database is also segmented into three fragments. Whenever a worker finishes a search assignment, it checks with the master to receive offset information for previously completed queries. If such information arrives, the worker will first write local qualified output blocks to the shared file system before searching its next assignment. Note that as the results merging cannot be finalized until all workers complete searching the query sequence q_i, a worker likely will not be able to proceed with output right after it finishes searching this query. Instead of blocking this worker until the write offsets for q_i are released by the master, the scheduler let it go ahead to request the next query sequence, q_{i+1} and start computation again.

The writing of non-contiguous output data can be done in two ways. The intuitive way is to perform a seek-and-write operation for every block via POSIX I/O calls. This is a slow solution as it will result in many small I/O requests, unfavored by typical file systems. An alternative way is to use the non-contiguous write method provided by MPI-IO [57]. Each worker first creates a file view that describes the locations to be written, then just calls `MPI_File_write()` to issue writes of all output data at once. MPI-IO libraries such as ROMIO [50] provide optimizations for this kind of non-contiguous write with data sieving [58]. In our experiments, MPI-IO calls will be used when data sieving is supported by the underlying file system, otherwise we fall back to seek-and-write with POSIX functions.

The major advantage of WorkerIndividual is that it does not introduce any synchronization overhead in the I/O phase. Workers alternate between computation and I/O, without wasting time waiting for other workers. This strategy is expected to work efficiently if the non-contiguous writing performance is well sustained by the underlying file system. The disadvantage, however, is that the non-contiguous accesses may be inefficient.

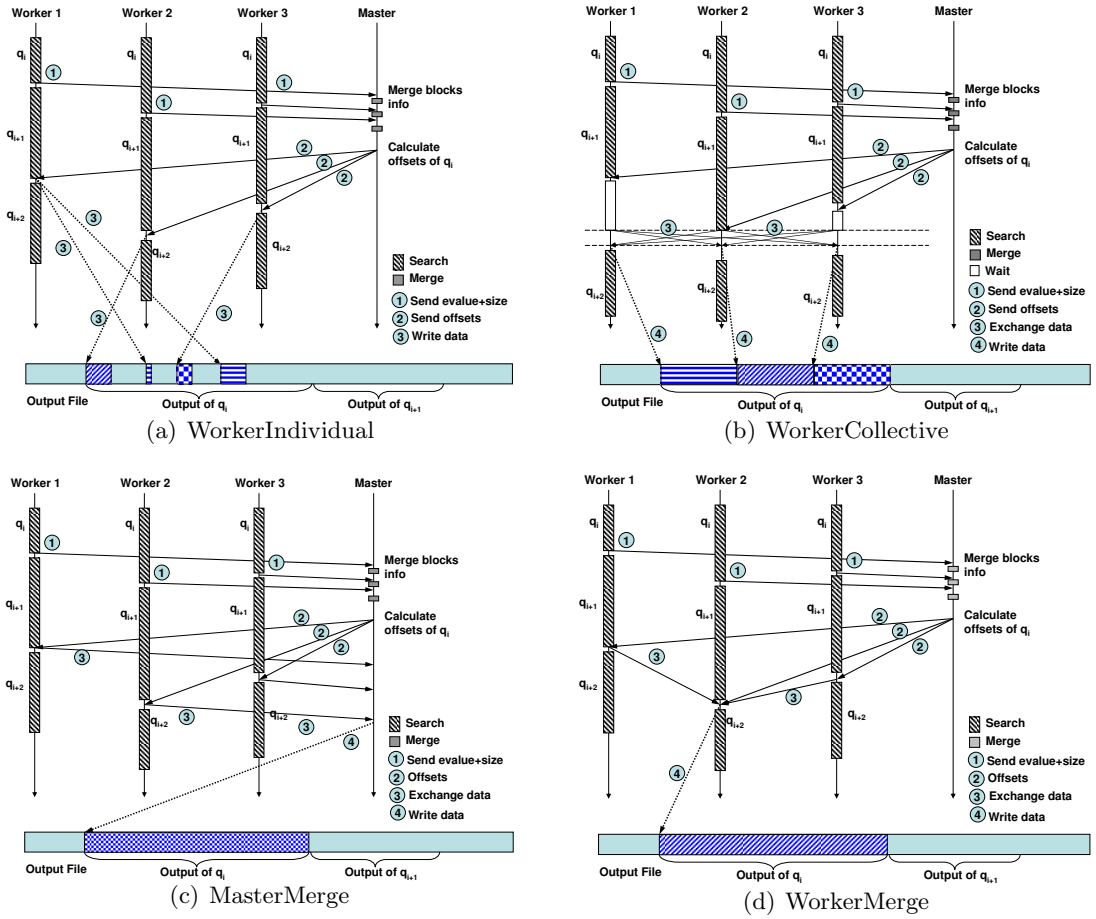


Figure 3.4: Four output strategies compared in this study. WorkerIndividual adopts the data sieving I/O technique when possible. WorkerCollective is based on the collective I/O technique. MasterMerge merges and writes output at the master. WorkerMerge is based on the proposed asynchronous, two-phase I/O technique.

Even with data sieving, the concurrent irregular I/O requests from multiple workers would generate much contention on the file system, leading to undesirable I/O performance.

WorkerCollective

Collective I/O appears to be a natural solution when we have a large number of small, non-contiguous I/O requests accessing a shared file with an interleaving pattern. A corresponding output strategy for parallel sequence search, which we call *WorkerCollective*, lets the workers coordinate their write efforts into larger requests. As illustrated in Figure 3.4(b), after receiving write offsets from the master, rather than performing individual writes, the workers will issue a MPI-IO collective write request. Like in the case of WorkerIndividual, the results merging of q_i likely will not be done right after the search of this query is completed. To overlap the master's result processing with workers' searching, all the workers involved in searching q_i continue with query processing, until between assignments they found that the file offset information regarding q_i has arrived. At this point, a worker will enter the collective output call for q_i. The advantage of this strategy lies in its better I/O performance compared to the non-contiguous write approach, by combining many small write requests into several large contiguous ones through extra data exchange over the network. However, even with the overlap discussed above, this strategy still incurs frequent synchronization, as collective I/O calls are essentially barriers that force workers to wait for each other (as shown with the white boxes in Figure 3.4(b)). While very suitable for time-step simulations, this communication pattern is undesirable for parallel sequence-searches, which are known to have imbalanced computation intervals.

MasterMerge

Another intuitive solution, especially considering the popular use of NFS servers on commodity clusters, is to let the master handle all the output. We call this the *MasterMerge*. With MasterMerge, the workers proceed as in the previous two schemes, until the result merging outcome is communicated back to the workers. At this point, rather than writing qualified local output blocks to the shared file, the workers forward them to the master. The latter then merges the output data in its memory and issues large, sequential write requests to the output file. Figure 3.4(c) shows the output process using MasterMerge.

This approach avoids concurrent I/O by many workers on a system with limited parallel I/O support and merges small I/O requests without enforcing synchronization on workers. However, the scalability of this scheme is apparently limited on larger systems, as the master can easily become the bottleneck.

In implementing this strategy, the master's memory constraint has to be taken into account. We defined a maximum write buffer size (MBS) in the master to coordinate incremental output communication, similar to the scheme used in common 2PIO implementations [49]. That is, only MBS amount of data will be collected and written at each operation.

WorkerMerge

Recognizing the limitations of the aforementioned approaches, we propose WorkerMerge, an output strategy that performs asynchronous, two-phase writes with merged I/O requests. With this strategy, after the master finishes result merging for query q_i, it appoints one of the workers to be the writer for this query. To minimize data communication, we select the worker with the largest volume of qualified output data to play the writer role, who will collect and write the entire output for this query. The workers involved in searching q_i are notified about the output data distribution and the writer assignment, and send their output data for q_i to the writer. In the example depicted in Figure 3.4(d), worker 2 is selected as the merger for query q_i. After receiving output offsets, worker 1 and worker 3 send their output blocks to worker 2 using non-blocking MPI sends, then continue with the next search assignment. After worker 2 finishes searching query q_{i+1}, it receives output blocks sent by worker 1 and 3, then performs a contiguous write.

In our implementation, the same incremental communication strategy used in MasterMerge is adopted here to guard against buffer space shortage. Such data collection is conducted using non-blocking MPI communication to overlap with search computations. A writer checks the status of the collection between searching two assignments. Whenever the data is ready, it issues an individual write call to output a large chunk of data.

The WorkerMerge strategy takes advantage of collective I/O and removes the synchronization problem. Meanwhile, it resolves the bottleneck problem of MasterMerge by offloading output gathering and writing to workers. It seamlessly works with our dynamically load-balanced computation scheduling algorithm and allows a large number of workers

to be efficiently supervised by a master.

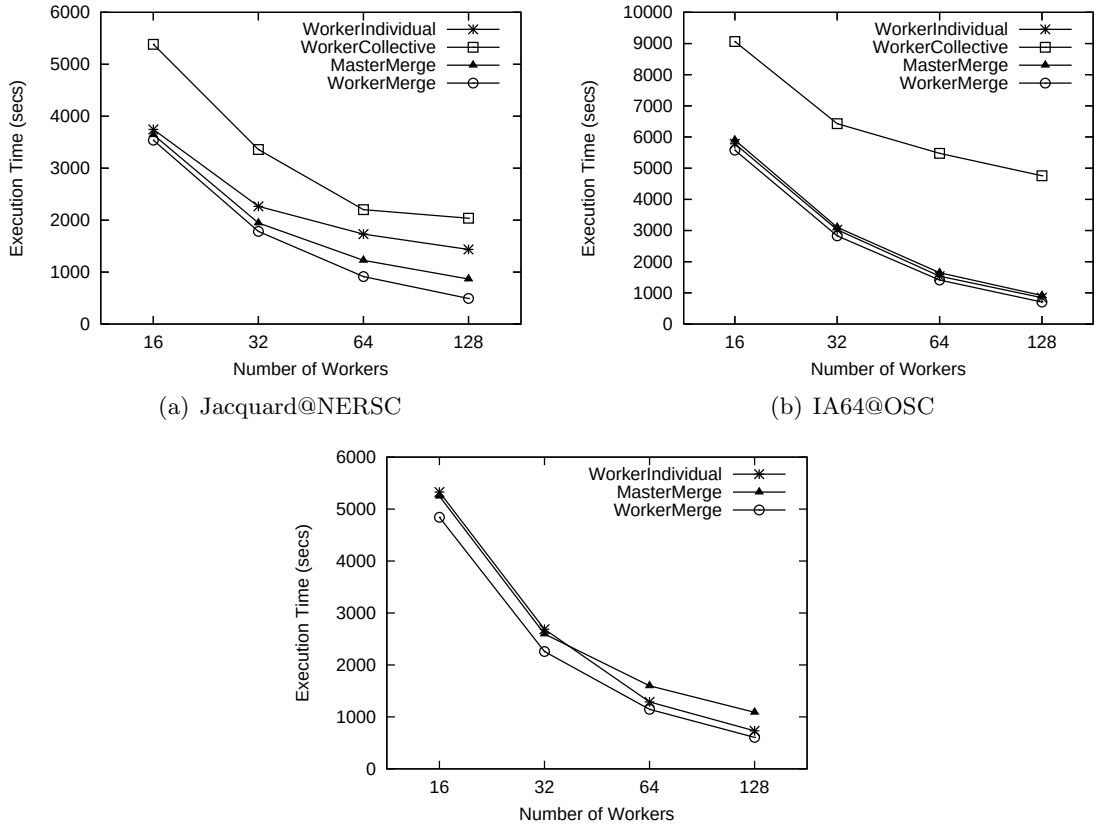
One may argue that the MPI-IO standard does provide asynchronous collective I/O with *split collective read/write operations* [57]. The split collective operations do allow the overlap of I/O and computation by separating a single blocking collective call into a pair of "begin" and "end" operations. However, our framework cannot benefit from them for two reasons. First, split collective I/O is not yet supported in popular MPI-IO libraries [50]. Second, in our target scenario, the data distribution (in terms of an MPI file view) is computed dynamically depending on the local result merging process, therefore a new file view needs to be constructed for each query's output. Since the MPI_File_set_view call has only a blocking form, there is no way to remove inter-worker synchronization even with split collective write functions.

In our current design, the result from each query is written by one writer process. For queries that generate large amounts of output data, using multiple writers may be beneficial. Our work targets large BLAST jobs processing many queries on supercomputers. With a large number of concurrent groups working on queries and our proposed asynchronous writing, the underlying I/O parallelism in the system is expected to be well utilized. Therefore the main issue here is whether the individual writers will have enough memory space to buffer the single-query output, which can be addressed by our incremental buffering and writing design.

3.4 Performance Evaluation

To evaluate the computation and I/O scheduling approaches presented earlier in this paper, we performed extensive experiments with mpiBLAST-PIO on three clusters with varying sizes, architectures, interconnection types, operating systems, and file systems, whose details will be presented in Section 3.4.1.

To reduce the test space, we first compare the four output strategies presented in Section 3.3.4. Then we configure mpiBLAST-PIO to use the best output strategy in the rest of our experiments and examine our computational load-balancing design. Finally, we systematically evaluate the scalability of mpiBLAST-PIO on a supercomputer with thousands of processors.



(a) Jacquard@NERSC

(b) IA64@OSC

(c) System X@VT

Figure 3.5: Node scalability results of searching 1000 randomly sampled `nt` sequences on different number of workers.

3.4.1 Experiment Setup

Below we give the detailed configurations of our test platforms.

Jacquard: Jacquard is a 356-node Opteron cluster running Linux, located at National Energy Research Scientific Computing Center (NERSC). Each node has dual Opteron 2.2 GHz processors and 6 GB of physical memory. The nodes are interconnected with a high-speed InfiniBand network. Shared file storage is provided by the GPFS filesystem [59]. No local storage is available for applications on the compute nodes. The MPI library is MVAPICH version 0.9.5-mlx1.0.1.

IA64: IA64 is distributed/shared memory hybrid of commodity systems based on the Intel Itanium 2 processor. It is located at Ohio Supercomputer Center (OSC). The

cluster is built using HP zx6000 workstations, an SGI Altix 3000 and several Altix 350s. The partition used in our experiments consists of 110 compute nodes, each has two 1.3 Gigahertz Intel Itanium 2 processors and 4 GB of physical memory. The nodes are interconnected with Myrinet and Gigabit Ethernet. Shared file storage is provided by a PVFS filesystem. A 36 Gigabytes, ultra-wide SCSI hard drive is attached to each node as local storage. The operating system is Linux and the MPI library is a version of MPICH optimized for the Myrinet high-speed interconnect.

System X: System X is a 1100-node MAC OS cluster located at Virginia Tech (VT). Each node consists of two 2.0-GHz IBM PowerPC 970 CPUs and 4 GB of physical memory. System X uses two interconnection fabrics, InfiniBand and Gigabit Ethernet. Shared file storage is provided by a ZFS [60] distributed filesystem. A 80GB hard drive is attached to each node as scratch space. The MPI library is a customized version of MPICH 1.2.5.

It worths noting that due to a special combination of the file system (ZFS) and the MPICH customization, MPI-IO (especially collective I/O) is not well supported on System X. Therefore for all experiments on System X, we only show results of three output strategies other than WorkerCollective.

The experiment database is `nt`, a nucleotide sequence database that contains the GenBank, EMB L, D, and PDB sequences. At the time when our experiments are performed, the `nt` database contained 5,454,516 sequences with a total raw size of about 20GB and a formatted size about 6.5GB. To stress test the scalability of mpiBLAST-PIO, we use sequences randomly sampled from `nt` itself as queries because these queries are guaranteed to find close matches in the database.

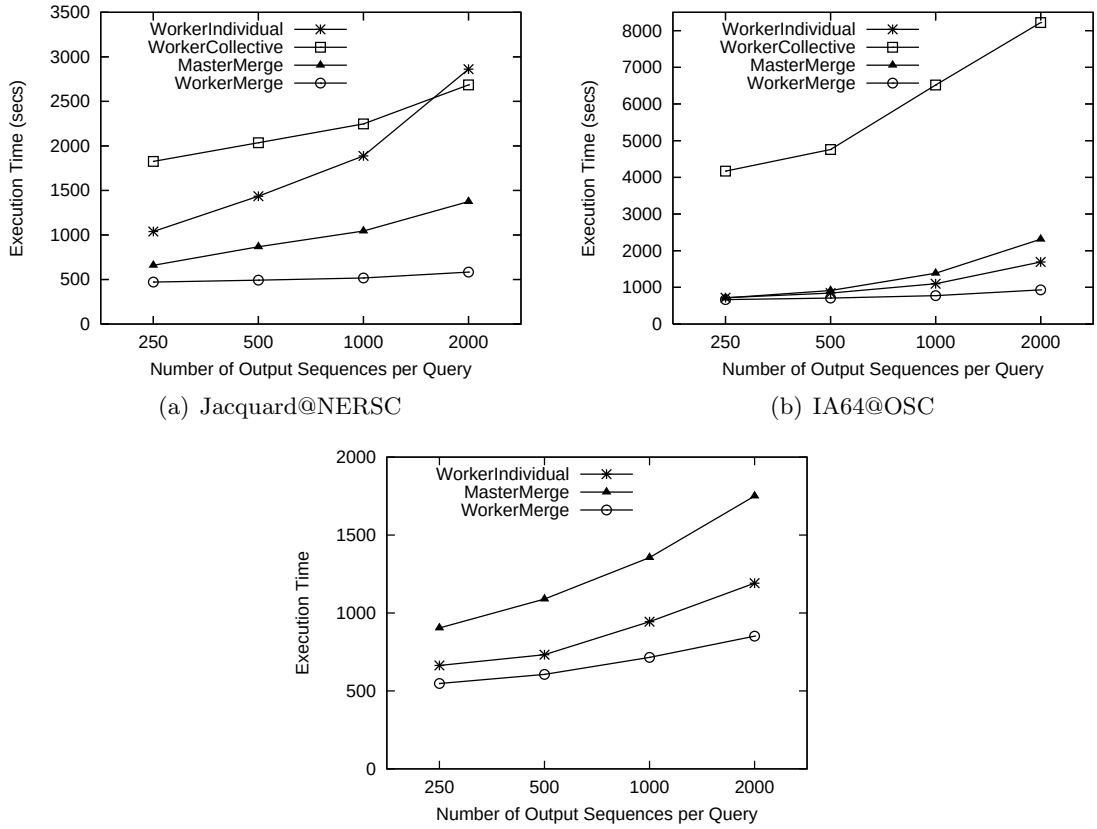
In our experiments, mpiBLAST-PIO is configured to run on the sharing mode on IA64 and System X, where the database is predistributed to the local disks of compute nodes to save the database redistribution time in consecutive runs. The execution times reported on these systems do not include the database distribution time. On Jacquard, the program is configured to run on the non-sharing mode as this platform does not provide per node local storage for applications. The sequence database is distributed to the memory of all processors using the replication approach described in Section 3.3.1 at each run, and this overhead is included in the overall execution time.

3.4.2 Scalability Comparison of Output Strategies

In this section we evaluate the scalability of four output strategies discussed in Section 3.3.4 with regard to both system sizes and output sizes. The experiment query set consists of 1000 randomly sampled `nt` sequences sized 5KB or less. The sequences within this length range account for 96% of overall sequences in the database. Our past experiences suggest that searching these sequences incurs high I/O demands. For all experiments in this section, we configured mpiBLAST-PIO to use only one partition. In doing so, we focus solely on I/O scalability and isolate other factors such as load balancing between partitions. The `nt` database is partitioned into 32 fragments. These fragments are distributed on every 8 workers in a round-robin fashion. This configuration works well for the BLAST search jobs used in the experiments according to our experiences. All experiments are repeated three times and the average results are reported. The result variances are less than 5% in these experiments, hence the error bars are not included in the figures.

Figure 3.5 shows the node scalability test results, where we plot the overall execution time of searching the given query set against `nt` as a function of the number of workers. We find that across all three platforms, the WorkerMerge approach works consistently the best. Overall, its winning margin increases as the number of workers grows. With 128 workers, it outperforms the WorkerInidividual strategy by an average factor of 1.8 over the three test systems, WorkerCollective by 5.4, and MasterMerge by 1.7. In addition, WorkerMerge achieves near-linear scaling from 16 to 128 workers on all three tested platforms. As expected, it outperforms other strategies by adopting distributed, merged I/O without enforcing additional synchronization that slows down query processing.

For the two systems that have collective I/O support, the WorkerCollective approach gives the worst performance. This is due to the synchronization cost associated with periodic collective I/O operations. Interestingly, the differences between the other three approaches look very different on the three platforms. Most notably, the WorkerIndividual, MasterMerge, and WorkerMerge strategies yield very similar performance on the IA64 system. One major reason is that the CPU frequency on this machine is relatively low (1.3GHz), while the interconnection networks and the I/O subsystem are with fairly high configurations. Overall this results in a lower pressure on the output components, as results are generated rather slowly but consumed fast. On Jacquard, it is evident that



(a) Jacquard@NERSC

(b) IA64@OSC

(c) System X@VT

Figure 3.6: Output scalability results of searching 1000 randomly sampled `nt` sequences with different amount of output data.

WorkerMerge outperforms MasterMerge, and MasterMerge outperforms WorkerIndividual.

Next, we perform a set of output scalability tests. By default, NCBI BLAST reports the matches between a query sequence and the top 500 database sequences that are closed to the query based on the alignment results. This configuration is used in the previous group of tests. In the output scalability tests, we vary the output size by configuring BLAST to report the top 250, 500, 1000, and 2000 result database sequences. The corresponding total output sizes are 428MB, 768MB, 1.4GB, and 2.4GB, respectively. The total number of workers used here is fixed at 128. The rationale behind the output scalability tests is that as parallel computers become more powerful and databases grow larger, the I/O-to-computation ratio of genomic sequence-searches is expected to increase in the near future.

By varying the amount of output data, our tests arguably evaluate how well different output strategies accommodate the performance trend of future sequence-search jobs. In addition, our tests address users' needs for gathering large amounts of results; according to the feedbacks provided in mpiBLAST users' mailing list, it is not unusual that nowadays BLAST users choose to have several thousands of result sequences reported.

Figure 3.6 shows the results of the output scalability tests, where we plot the overall execution time as a function of the number of reported result sequences. The advantage of WorkerMerge over the other strategies is more evident than in the node scalability tests. When reporting 2000 result sequences, WorkerMerge outperforms the second best strategy by a factor of 2.4, 1.8 and 1.3 on Jacquard, IA64 and System X respectively, and outperforms the worst strategy by a factor of 4.9, 8.8, and 2.1. In addition, the performance curves of WorkerMerge are much flatter than those of the other three strategies, suggesting that WorkerMerge is less sensitive to the growth of output sizes than the others.

Overall the relative performance differences between various strategies are similar to those in the node scalability tests. A new observation is that on Jacquard, the performance of WorkerIndividual degrades fast as more results are reported, causing a worse execution time than WorkerCollective at 2000 result database sequences. The scalability of WorkerIndividual is much better on IA64 and System X. One explanation is that the non-contiguous write approach used in WorkerIndividual is better supported by the file systems on the two platforms. In particular, PVFS (on IA64) provides special optimization for this write pattern with LIST I/O technique [52].

It is clear that for both types of scalability tests, WorkerMerge greatly outperforms the other strategies on all three tested platforms. Therefore we will configure mpiBLAST-PIO to use WorkerMerge in the rest of experiments.

3.4.3 Fine-grained Dynamically Load Balancing

In Section 3.3.2 we presented the details of our fine-grained, dynamically load-balancing algorithm. A key factor of the design is to minimize the scheduling overhead by having masters proactively prefetch query segments from the supermaster. In this section, we evaluate the efficacy of query-segment prefetching with two synthesized workloads that have different balancing sensitivity to the task granularity.

The first workload uses a query set consisting of 1000 randomly sampled `nt` se-

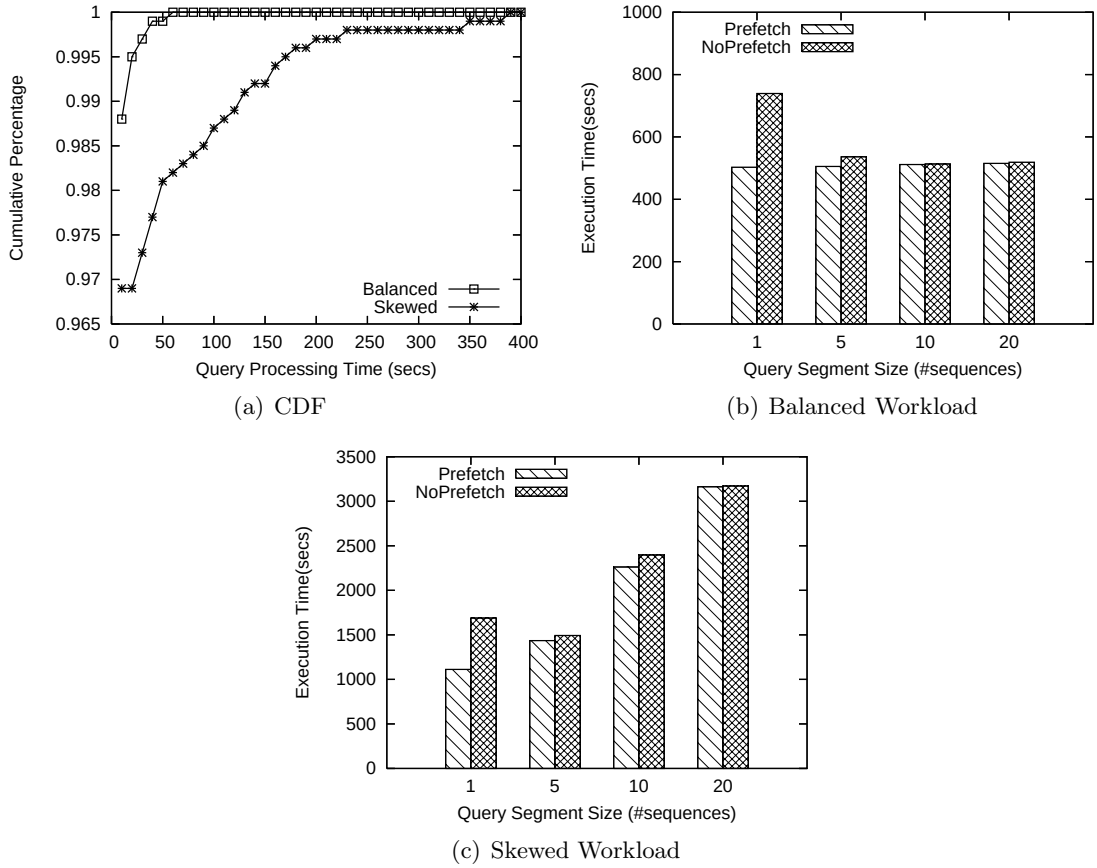
quences sized 1KB or less. Our past experiences suggest that the search time distribution is relatively balanced for these sequences. For this workload, the load balance results are relatively insensitive to the task granules. For the second workload, we purposely synthesize a query set with a skewed search time distribution, where the load balancing results are highly sensitive to the task granules. We mixed expensive queries with inexpensive ones in terms of their search times as follows. According to reference [38], expensive query sequences in the `nt` database are likely to be larger than 5KB. With this hint, we first randomly sample 10,000 query sequences under 50KB from the `nt` database. These sequences are separated into two groups, with the first group consisting of sequences larger than 5KB and the second group consisting of the rest. Then we extract 10% of the most expensive sequences out of group one and put them together with 10% sequences randomly extracted from group two. This results in a 1000-sequence query set with expensive ones in the beginning.

Figure 3.7(a) gives the CDF function of the processing time of each query sequence in the above two workloads when searched on 32 processors on System X. We label the first and the second workloads with "Balanced" and "Skewed" respectively. As can be seen, 98.8% of query sequences in the first workload can be processed within 10 seconds, and the longest processing time is only 50 seconds. In the second workload, most of the query sequences (96.9%) can still be processed within 10 seconds. However, the rest of the sequences are much more expensive, with processing times ranging from 40 to 400 seconds.

We search the two workloads on System X using 166 processors configured into 5 partitions, with each consisting of 32 workers. The `nt` database is partitioned and predistributed in the same way as in the scalability tests. Each experiment is repeated three times and the average numbers are reported. Again, the result variance are quite small (less than 5%) so we do not include the error bars in the result figures.

Figure 3.7(b) shows the results of searching the balanced workload with and without query-segment prefetching on various task granules (i.e. query segment sizes). The overall execution time is reported as it measures the comprehensive performance impacts and matters the most to the end users. As expected, our prefetching design can significantly reduce the scheduling overhead when using small task granules compared to the non-prefetching design. Specifically, without prefetching of query segments, the overall execution time increases by a factor of 1.4 when the query segment size drops from 5 to 1. This is because when the task granule is small, there is no enough work in a query segment

to balance loads across workers in the partition, causing worker idleness. In contrast, with prefetching, the overall execution time is about the same across different segment sizes. This suggests that our query prefetching design can gracefully hide the scheduling overhead of doing fine-grained load balancing in the given setting.



(a) CDF

(b) Balanced Workload

(c) Skewed Workload

Figure 3.7: Performance impacts of query segment prefetching.

The results of searching the skewed workload are quite different than those of the balanced workload, as shown in Figure 3.7(c). For this heavy-headed query set, a large query segment size can cause significant imbalance in the processing times of individual segments. As a result, in general the overall execution time increases as the query segment size grows. However, without prefetching, using segment size 5 delivers better performance than using segment size 1. The reason is when the segment size is 1, the overhead caused

by the worker idleness offsets the gain of fine-grained load balancing. The worker idle issue is greatly resolved with prefetching of query segments. Consequently, with prefetching, the system achieves the best performance at the smallest task granule (query segment size 1), where the advantage of fine-grained load balancing is fully taken. In particular, the best prefetching case (query segment size 1) outperforms the best non-prefetching case (query segment size 5) by a factor of 1.4.

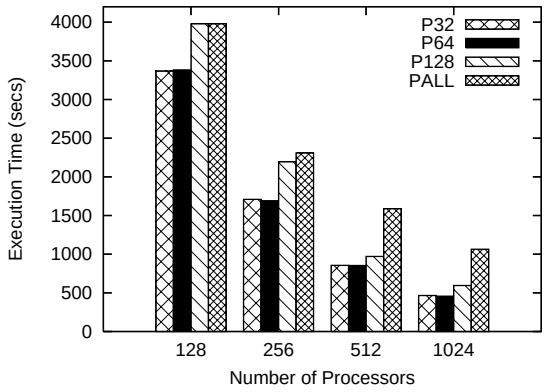


Figure 3.8: Scalability of searching 5000 `nt` sequences with various partition sizes on System X. Configurations of different partition sizes are labeled P32, P64 and P128 respectively. PALL refers to the single-layer master-slave configuration, using all available processors for one partition. Note P128 and PALL are actually the same when running on 128 processors.

3.4.4 Overall System Scalability

In order to evaluate the scalability of our integrated scheduling approach, in this experiment we benchmark mpiBLAST-PIO with up to 1024 processors on System X. We vary the system size to include from 128 to 1024 processors. To see how different partition sizes will affect the system throughput, for each system size, we perform multiple tests by configuring mpiBLAST-PIO to use 4 different partition sizes (in terms of number of workers): 32, 64, 128 and the system size. Note when the partition size is set to the system size (e.g., using 128 as the partition size on 128 processors), mpiBLAST-PIO acts in a single-layer master-slave mode, where all the workers in the system are overseen by just one master process. The query set consists of 5000 randomly sampled `nt` sequences sized 5KB or less. The database is partitioned and distributed as other experiments presented

previously. The prefetched query segment size is set to one.

The performance results are shown in Figure 3.8. The first observation is that the execution times decrease nicely for all configurations as the system size grows. Surprisingly, even with a single-layer configuration (labeled PALL in the figure), mpiBLAST-PIO scales well to 1024 processors, which suggests that our inner-partition scheduling algorithm is highly efficient. However, the benefit of hierarchical scheduling on large system size is evident. With 512 processors and above, using single-layer scheduling is significantly slower than all other three configurations, simply because the master will become a performance bottleneck when managing too many workers. Specifically, PALL is slower than the best hierarchical configuration (P64) by a factor of 1.9 and 2.3 on 512 and 1024 processors respectively.

The performance impacts of using different partition sizes are determined by a combination of several factors. On the one hand, using larger partition sizes can save the number of master processes and give more horse power to the actual search computation. On the other hand, larger partition sizes could overburden the master process with increasing loads of scheduling and output coordinating, and consequently incur higher parallel overhead. As can be seen in Figure 3.8, for this particular setting, using 128-worker partitions (labeled P128) is noticeably slower than the other two configurations (P32 and P64), mainly because the master is overloaded when handling a large number of workers. Interestingly, P32 and P64 deliver almost identical performance. This suggests that when partition size increases from 32 to 64, the saving in search computation time is counteracted by the parallel overhead increased. Nonetheless, mpiBLAST-PIO achieves almost linear speedup up to 1024 processors when the partition size is set to 64, with a parallel efficiency of 92% on 1024 processors as shown in Figure 3.9.

3.5 Case Study on IBM Blue Gene/P Supercomputer

In this section, we present a case study of using mpiBLAST-PIO to solve a sequence-search problem in the real world on a IBM Blue Gene/P platform with *tens of thousands* of processing cores.

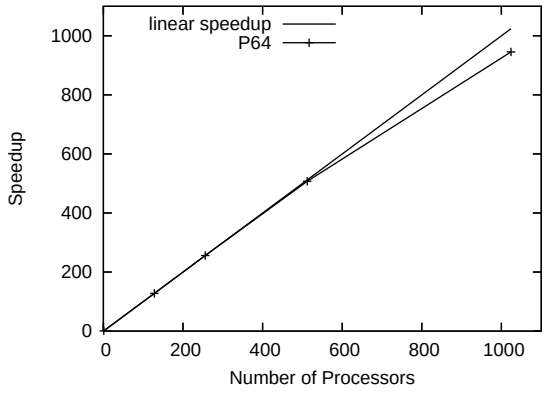


Figure 3.9: Speedup of searching 5000 `nt` sequences on System X with hierarchical scheduling (partition size 64).

3.5.1 Overview of the Blue Gene/P Architecture

The Blue Gene/P architecture supports a distributed memory, message-passing programming model [61]. It uses system-on-a-chip (SoC) technology to deliver four 850-MHz PowerPC 450 processors, capable of achieving a theoretical peak performance of 13.6 gigaflops/chip [62]. Each such SoC constitutes a Compute Node. A Compute Node attached to a processor card with 2 GB of memory creates the compute and I/O cards. Two rows of 16 compute cards then make up a node card. Next, a midplane consists of 16 node cards stacked in a rack. A rack holds two midplanes for a total of 32 node cards.

The PowerPC 450 core itself contains the first-level (L1) cache, which is 64-way set associative. The second level (L2R and L2W) of caches, one dedicated per core, are 2 KB in size. They are fully associative and coherent; they act as prefetch and write-back buffers for L1 data. The L2 cache line is 128 bytes in size. Each L2 cache has one connection toward the L1 instruction cache running at full processor frequency. Each L2 cache also has two connections toward the L1 data cache, one for the writes and one for the loads, each running at full processor frequency. The third-level (L3) cache is 8-way set associative and 8 MB in size with 128-byte lines. Both banks can be accessed by all processor cores. The L3 cache has three write queues and three read queues: one for each processor core and one for the 10-Gigabit network.

There can be up to two I/O cards per node card. When these nodes do not have

a local file system, I/O operations need to be sent to an external device. In order to reach this external device (outside the environment), a *compute node* sends data to an I/O node, which in turn carries out the I/O requests [62]. In the BG/P systems used in our study, the file system are configured with the Global Parallel File System (GPFS) [59, 61].

Applications on Blue Gene/P may run in three different modes: Symmetrical MultiProcessing (SMP) Node mode, Virtual Node mode (VN), and Dual Node mode (DUAL). In the first mode, each compute node executes a single task with a maximum of four threads. Node resources (primarily the memory and the torus network) are shared by all threads. In VN mode, four single-threaded tasks are run on each node, one task per core. Each task gets 1/4 of the total memory of the node. Finally, in the DUAL mode, two tasks can be run on a node. Each task gets half of the memory and cores and can consist of at most two threads.

3.5.2 Problem Description

In this case study, we sequence search the entire microbial genome database against itself, which has several major utilities in computational biology as described in [13]. A summarized description of these utilities is noted below:

Discovering Missing Genes: Genome annotation identifies the location of genes and the coding regions in a genome, determines what those genes do, and then annotates this information to the genome. Part of the above process entails accurately determining the location and structure of protein-encoding and RNA-encoding genes via computational analysis. If done improperly, we end up *predicting false genes* or *missing real genes*.

A popular method for locating genes, known as the similarity method, requires the comparison of genomic segments with a database of gene sequences found in similar organisms. If the sequence is conserved, then the segment that is being evaluated is likely to be a coding gene. Genes that do not fit a given genomic pattern and do not have similar sequences in current annotation databases may be systemically missed.

To detect missed genes, we use the similarity method and compare raw genomes against each other rather than comparing a raw genome to a database of known genes. For instance, if gene x in genome X and gene y in genome Y have been missed and x is similar to y, then the similarity method will find both. However, the *only* way of identifying this is to perform an all-to-all comparison of the entire microbial genome database against itself,

which is highly compute-intensive.[2]

Adding Structure to Genetic Sequence Databases: One of the major issues with sequence searching is the structure of the sequence database itself. Currently, these databases are "structured" as a flat file in human-readable format, e.g., ASCII text, and each new sequence that is discovered is simply appended to the end of the file. Without more intelligent structuring, the query sequence needs to be compared to every sequence in the database (several millions currently) forcing the best-case to take just as long as the worst-case. By organizing and providing structure to the database, searches can be performed more efficiently by being able to discard irrelevant portions entirely. One way to provide structure to the sequence database is to create a sequence similarity tree where "similar" sequences are closer together in the tree than dissimilar sequences. The connections in the tree are created by determining how "similar" the sequences are to each other; sequence search can be used to determine the sequence similarity. To create *every* connection, however, an "all-to-all" sequence search must be performed where the input query being the same as the database, resulting in an output size of N^2 values (where N is the number of sequences in the database).

3.5.3 Performance Results

At the time when the experiments were carried out, the microbial sequence database contains 16 million sequences with an approximately 6 GB raw size. We first benchmarked the scalability of mpiBLAST-PIO on Blue Gene/P by searching 0.25 million query sequences randomly sampled from the microbial genome database against the database itself. We performed initial profiling by varying the partition size, the number of database replicas per partition, and the number of database fragments. This profiling allowed us to identify the ideal values for different parameters for our system — 64 database fragments, 128 as the partition size, and 16 as the replication group size.

Based on this profiled information, in this experiment, we increased the system size from 1 rack to 8 racks and measured the speedup achieved as illustrated in Figure 3.10. As can be seen, mpiBLAST-PIO scales almost linearly from 4096 cores to 32768 cores, achieving a *93% parallel efficiency* at the largest test scale. This near-perfect speedup demonstrates

[2]When this computation was done as part of the SC—07 Storage Challenge, it required 12,000+ processors to compute over a two-week period and write to a petabyte file system.

that the synergy of mpiBLAST-PIO's scalable task and I/O scheduling design can take full advantage of the massive parallelism offered by the BG/P system.

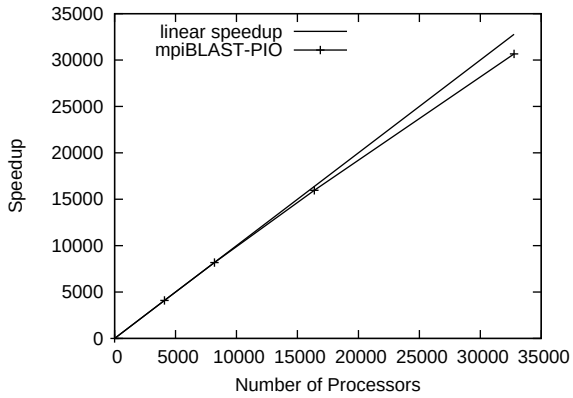


Figure 3.10: Speedup of searching 0.25 million of microbial genome sequences against the microbial genome database itself.

Next, we leveraged the processing power of BG/P system, enhanced by our optimizations, to solve the problem of searching the entire microbial genome against itself. To avoid data loss of possible hardware failures during the long run, we split the database into 64 query files, each consisting of about 85MB of sequence data, and continue submitting jobs to the 8-rack system until the whole genome search is completed. This problem, previously considered to be computationally intractable in practice, was completed within *12 hours*.

Chapter 4

Semantic-based Distributed I/O with the ParaMEDIC Framework

4.1 Motivation

Nowadays the compute and storage resource requirements of sequence-search jobs in advanced computational biology are greater than ever before. Most life scientists do not have the local access to the supercomputing resources needed for their search jobs. Thus the sequence-search results generated on the remote supercomputers need to be frequently offloaded to local for visualization and further analysis. Moreover, highly resource demanding search jobs such as database-to-database alignments may require thousands to tens of thousands of processors for timely response and generate hundreds of terabytes of results data. For those jobs, few supercomputing sites are equipped with both the necessary compute and storage resources. Thus distributed sequence searching across multiple supercomputing cites is an effective approach to address the resource provisioning challenge. For example, a large sequence-search job can be split and shared across several compute-resource-abundant sites, and the results data can be write to a storage-resource-abundant site.

To facilitate large-scale collaborative scientific computing, people have invested in high-speed distributed network connectivity for moving massive data across different sites. However, these high-speed networks do not provide end-to-end connectivity to a very high percentage of scientific community. Moreover, the results data generated by advanced sequence-search jobs can be so large that the execution time is dominated by the distributed

I/O time even when the network bandwidth is fully utilized. To address this issue, in this work we present "Para-MEDIC: Parallel Metadata Environment for Distributed I/O and Computing", which leverages the application-specific meta-level information as well as effectively utilizes both local and remote resources to achieve efficient data movements in distribute environments. Specifically, ParaMEDIC first transforms the results data generated on the compute site into orders-of-magnitude smaller application-specific *metadata*. These metadata is then transfered to the remote storage cite and transformed back to the original results data. Essentially, ParaMEDIC trades a small amount of additional computation for significant reduction in the data volume to be transfered across the network.

ParaMEDIC can be viewed as meta-level data compression at a high level. Instead of perceiving the data as a generic byte-stream, ParaMEDIC allows application developers to plug in application-specific *knowledge* to achieve a much higher compression ratio. Consequently, it loses some level of portability compared to general data compression. In this work we demonstrate that sequence search can greatly benefit from such a tradeoff between portability and compression effectiveness, and we believe this approach can be applied to a broad class of meta information rich scientific applications such as visualizations.

We integrated ParaMEDIC with mpiBLAST-PIO (mpiBLAST thereafter) and evaluated the efficacy of the ParaMEDIC framework on several distributed environments. We first built a controlled environment on a local cluster that allows varying the bandwidth and latency between any two of individual nodes. This environment enables us to study how the performance of ParaMEDIC is affected under different network configurations. We then deployed ParaMEDIC in two real-world distributed systems, including 1) a slice of the TeraGrid consisting of nodes at the University of Chicago and San Diego Supercomputing Center, and 2) a system with a secure filesystem hosted between the Argonne National Laboratory and Virginia Tech over an Internet2 connection. Our experimental results demonstrated that ParaMEDIC can deliver *order-of-magnitude* performance improvements compared to the traditional approach of moving data as an opaque stream in distributed environments.

4.2 Distributed Environments

A wide variety of distributed environments exist today, ranging from high-latency, high-bandwidth *LambdaGrids*, to low-bandwidth environments connected over the Internet,

to unsecure environments requiring data encryption before transmission. Here we present two sample distributed environments.

4.2.1 NSF TeraGrid

NSF TeraGrid is a distributed computing facility that combines leadership-class resources at 11 partner sites within the U.S. to form the world's largest distributed cyber-infra-structure for open scientific research. It includes 750+ teraflops of computing capability and 30+ petabytes of data storage, along with rapid access and retrieval over high-performance networks to form the world's largest distributed cyber-infrastructure for open scientific research.

The TeraGrid comprises several sites including the University of Chicago/Argonne National Laboratory (Illinois), San Diego Supercomputing Center (California), Purdue University (Indiana), Texas Advanced Computing Center (Texas), and others. The San Diego Supercomputing Center (SDSC) also doubles as a host for a global parallel filesystem (GPFS) that is visible and usable by all TeraGrid compute servers. All sites are connected using high-bandwidth optical links. However, the large physical distance between the sites forces the latency to be high (up to tens of milliseconds) as well.

Scientists use the computational power available at different locations to run computations and then write the final output to the globally shared filesystem to be viewed or post-processed at a later time. While such a system provides good computational capability for I/O-rich applications, the distributed filesystem can form a significant bottleneck.

4.2.2 Argonne-VT Distributed System

The Argonne-VT distributed system is a small-scale research infrastructure built to share application output for post-processing and visualization. The system consists of 200 processors of shared compute resources and about 200 gigabytes (GB) of main memory spread across the two sites. In addition, the Argonne site provides 10 terabytes (TB) of storage resources, which are shared across the two sites using a distributed filesystem over a shared Internet2 connection (1 Gbps). This network connectivity is *much* slower than the NSF TeraGrid infrastructure. Furthermore, though Internet2 is mostly a dedicated infrastructure due to its relatively low utilization, it occasionally experiences large traffic bursts,

causing performance degradation in the network. Finally, because the connection between the two sites is over the traditional wide-area network, it is considered to be unsecure, and thus, the data transmission might require encryption, which can add substantial overhead as well.

4.3 The Design of ParaMEDIC

In this section, we present the detailed design of the Para-MEDIC framework (short for *Para*llel *M*etadata *E*nviron-ment for *D*istributed *I*/O and *C*omputation).

4.3.1 The ParaMEDIC Framework

The architecture of ParaMEDIC is shown in Figure 4.1. In summary, ParaMEDIC provides three major components that allow applications to take advantages of the meta-level data transferring:

- *ParaMEDIC Data Tools.* The data tools offer efficient supports of data-touching services such as data encryption and integrity check. These services are necessary when moving privacy sensitive data over unsecure distributed environments.
- *Communication Services.* The communication services handle data transferring over various networking protocols (e.g., TCP).
- *Application Plugins.* Applications plugins allow applications to inform ParaMEDIC how to create and process the semantic based metadata.

An application plugin should provide two types of functionalities. First, on the computing site, the plugin tells ParaMEDIC how to transform the application's results data into semantic-based metadata. Depending on the applications, the metadata can be converted directly from the application's intermediate results representation or extracted out from the actual output written to the disks. Second, on the storage site, the plugin provides a mechanism to convert the metadata generated at the computing site back to the original output.

Since the writing of application-specific plugins requires intimate knowledge of applications, currently ParaMEDIC relies on the application writers for plugin development

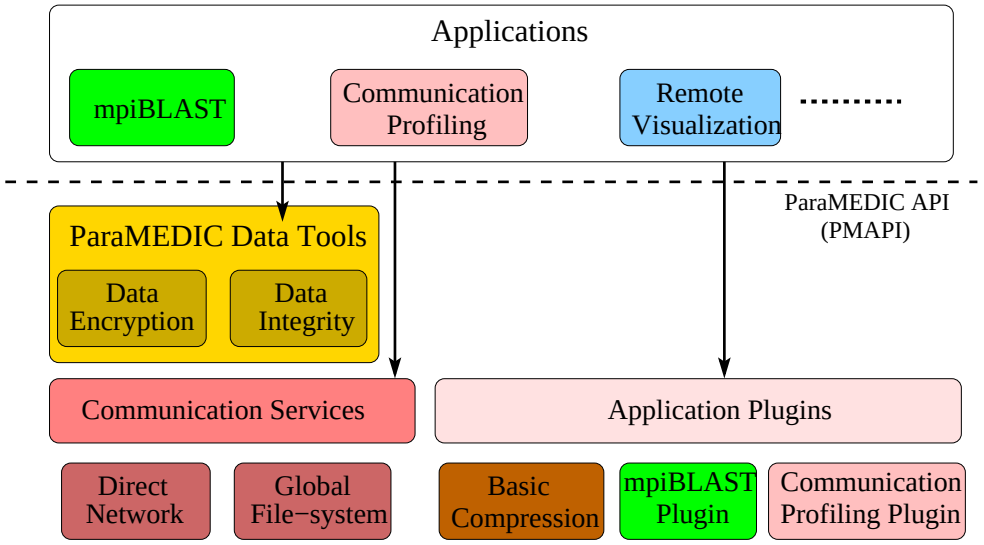


Figure 4.1: ParaMEDIC Architecture

but provides auxiliary tools to ease the task. In practice the plugin development is mostly straightforward. ParaMEDIC also provides a plugin that implements standard byte-stream oriented data compression and decompression. This plugin can be used for some part of the results data whose format can sufficiently benefit from the standard data compression.

4.3.2 Trading Computation with I/O Cost

The ParaMEDIC framework strives to trade a small amount of additional computation for reduction in the I/O data volume. The additional computation cost comes from post-processing the results data to generate metadata on the compute site and transforming the metadata back to the final output on the storage site. Apparently, not all applications can benefit from this approach. To complicate the issue even more, the cost of data post-processing and metadata conversion can be tuned relative to the data reduction ratio. In general, more post-processing computation can lead to better size reduction for the metadata. In this section we formalize the applicability of the ParaMEDIC framework in regard to the tradeoff between the additional computation cost and the I/O cost saving.

We consider a distributed environment consisting of a compute site and a storage site connected with wide area networks. We first introduce the following notations.

B: Network bandwidth in the above distributed environment.

T: Overall application computation time.

D: Total data generated by the application.

f(x): Time taken to convert output data to x units of metadata.

g(y): Time taken to convert y units of metadata to final output.

Within the ParaMEDIC framework, the overall execution time of an application (A) includes the application computation time, the metadata generating time, the metadata transferring time and the output regenerating time, which can be represented as the following equation based on the above notations.

$$A = T + f(x) + \frac{x}{B} + g(x) \tag{4.1}$$

.

In contrast, the execution time of the same application without employing the ParaMEDIC framework is simply the sum of its computation time and the distributed I/O time of the overall results data.

$$A' = T + \frac{D}{B} \tag{4.2}$$

Clearly, ParaMEDIC is only beneficial when A (equation 4.1) is small than A' (equation 4.2). That is,

$$T + f(x) + \frac{x}{B} + g(x) < T + \frac{D}{B}$$

Simplifying the above equation, we get:

$$D - x > B \times (f(x) + g(x)) \tag{4.3}$$

According to Equation 4.3, ParaMEDIC would save the overall execution time only when the size difference between the original data and the metadata is larger than the product of the network bandwidth and the data processing cost (i.e., the cost of generating metadata and converting it back to the original output data). Consequently, we expect ParaMEDIC to effectively save the overall computation time when the network bandwidth is low or the data processing cost is low. One important implication is that ParaMEDIC can be applicable even in the distributed environments with high network bandwidth if an

application's data processing cost is low enough.

4.4 Integration mpiBLAST with ParaMEDIC

In a *cluster* environment, most of the mpiBLAST execution time is spent on the search itself, i.e., comparing input query sequences to the database fragments, because the search requires a full scan of the database fragment and the BLAST alignment algorithm is computationally intensive (quadratic complexity). In contrast, the cost of formatting and writing the results that are generated from the search is much less significant, especially when many advanced clusters are configured with high-performance parallel filesystems.

However, in *distributed* environments such as those presented in §4.2, the execution profile of mpiBLAST differs significantly from cluster environments because mpiBLAST output needs to be written over a wide-area network to a remote filesystem. Hence, the cost of writing the results can easily dominate the execution profile of mpiBLAST, and thus, become a severe performance bottleneck.

This is where "ParaMEDIC comes to the rescue" for mpiBLAST. By replacing the traditional global parallel filesystem over a wide-area network with the ParaMEDIC framework (as shown at the top of Figure 4.2), we can still support the basic functionality of a global parallel filesystem, e.g., transferring large volumes of data to a distant filesystem (if needed), but more importantly, we can also provide advanced functionality that trades a small amount of additional computation for a potentially significant reduction in data that needs to be transferred in distributed environments. For example, as we will see in §4.5, a mpiBLAST-specific instance of ParaMEDIC reduces the volume of data that needs to written across a wide-area network by *more than two orders of magnitude*.

Specifically, Figure 4.2 depicts how mpiBLAST can be integrated with the ParaMEDIC framework. First, on the compute site (the left cloud in Figure 4.2), instead of having mpiBLAST collect and write all the result sequences and their matches of a query sequence, the mpiBLAST application plugin in ParaMEDIC, as shown in Figure 4.1, generates semantics-based metadata based on the mpiBLAST output at the compute site. ParaMEDIC then transfers this metadata to the I/O site (the right cloud in Figure 4.2), metadata that is orders of magnitude smaller than the actual data output that would have been transferred in a traditional global parallel filesystem. Next, at the I/O site, a small amount of addi-

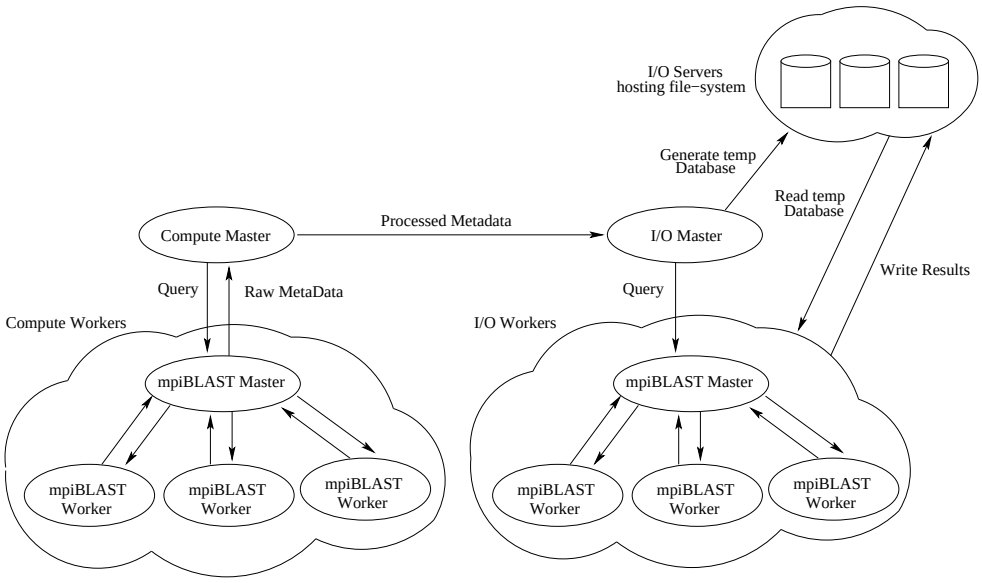


Figure 4.2: ParaMEDIC and mpiBLAST Integration

tional computation must then be performed on the metadata in order to re-generate the actual output data. That is, a temporary (and much smaller) database that contains only the result sequences is created by extracting the corresponding sequence data from a local database replica. ParaMEDIC then re-runs mpiBLAST at the I/O site by taking as input the same query sequence and the temporary database to generate and write output to the local filesystem. (Note: The overhead in re-running mpiBLAST at the I/O site is quite small as the temporary database that is searched is substantially smaller with only 500 sequences in it by default, as opposed to the several millions of sequences in large DNA databases.

4.5 Experimental Results

This section presents a performance evaluation of mpiBLAST in its native form, as compared to ParaMEDIC-enhanced mpiBLAST (hereafter referred to as simply ParaMEDIC). All experiments were performed with the Nucleotide (NT) database downloaded from the NCBI website. NT is a nucleotide sequence database that contains the GenBank, EMB L, D, and PDB sequences. At the time when our experiments were performed, it

contained over 5 million sequences with a total raw size of about 20GB. All queries were synthesized by randomly sampling sequences from NT itself.

4.5.1 Experimental Testbeds

Testbed 1 (Local Cluster): This testbed consists of 24 dual-2.8GHz-Opteron-processor dual-core nodes. Each processor has 2MB of L2-cache, and each node has 4GB of 667MHz DDR2 SDRAM and four SATA disks using a software RAID0. The nodes were connected with NetEffect NE010 10-Gigabit Ethernet adapters. We used NetEm to emulate the various distributed computing infrastructures. This allowed us to emulate high-latency, high-bandwidth distributed computing environments by separating the nodes in the system into two sub-clusters, where communication within the sub-cluster is fast but between sub-clusters is slow.

Testbed 2 (Argonne-VT Distributed System): This test-bed consists of two clusters (one at Argonne and one at VT) connected over Internet2. The Argonne cluster is the one described in *Testbed 1*, while the VT cluster consists of a 24-node Orion Multisystems DT-12 system that contains 12 individual x86 compute nodes in a 24" x 18" x 4" (or one cubic foot) pizza-box enclosure. Each compute node contains a Transmeta Efficeon processor, its own memory, and Gigabit Ethernet interface. The nodes share a power supply, cooling system, and external 10-Gigabit Ethernet network connection.

Testbed 3 (NSF TeraGrid): This testbed consists of a subset of the TeraGrid, using nodes at U. Chicago and SDSC. The U. Chicago site consists of two sets of nodes. The first set has 96 2.4GHz Intel Xeon 32-bit dual-processor systems, each with 3GB memory and 512KB L2 cache, while the second set has 64 1.5GHz Intel Itanium II 64-bit dual-processor systems, each with 4GB memory. The SDSC site has 64 1.5GHz Intel Itanium II 64-bit dual-processor systems, each with 4GB memory. The two sites are connected with a 30-Gbps high-bandwidth network, and the end-to-end delay between the two sites is approximately 10ms.

4.5.2 Local Cluster Evaluation

Here we compare ParaMEDIC to native mpiBLAST on a local cluster, which emulates various distributed infrastructures.

Impact of High-Latency, High-Bandwidth Networks

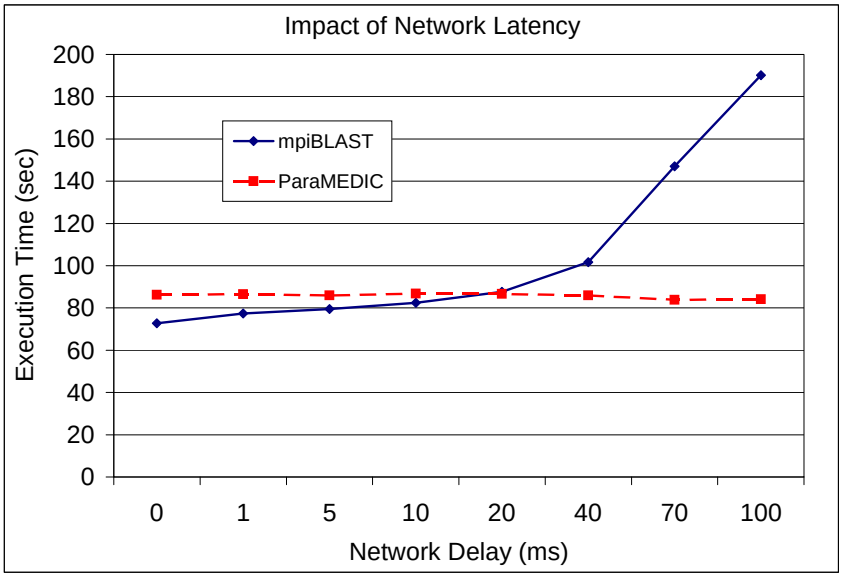
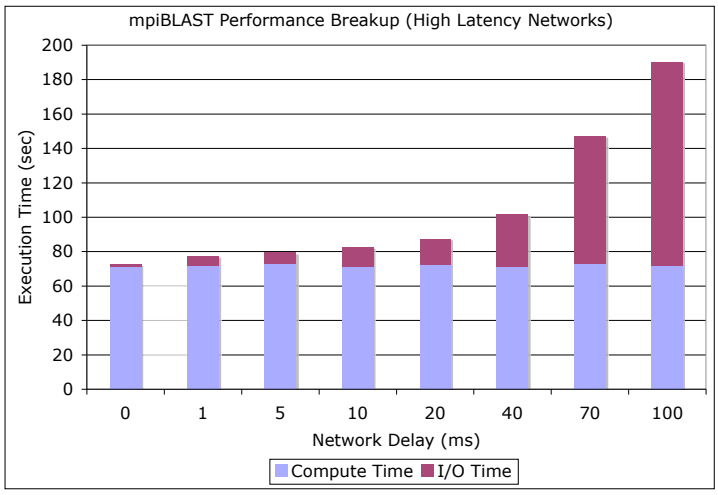


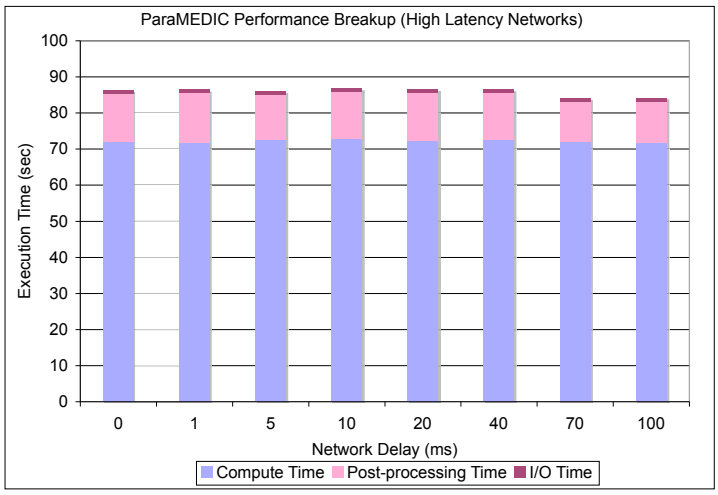
Figure 4.3: Impact of High Latency Networks

We analyze the impact of distributed environments connected with high-latency, high-bandwidth networks on the performance of ParaMEDIC and basic mpiBLAST. For this experiment, we divide the local cluster into two logical sub-clusters. While all the nodes are connected with a 10Gbps network, we artificially delay the communication between nodes belonging to different sub-clusters. The performance of the application for different network delays is measured (the amount of delay is fixed within a run and is illustrated on the x-axis). The socket buffer sizes are set to be equal to the bandwidth-delay product of the network so as to maximize the performance that the network subsystem can provide. Four nodes (each node with 4 SATA disks connected with a software RAID0) in the second sub-cluster host a PVFS2 filesystem which is visible to all nodes in both the sub-clusters.

For the evaluation, 80 processors are used for performing the computation. For mpiBLAST, all processors were used for performing the actual application computation. However, for ParaMEDIC, to keep the overall computational resources constant, 76 processors hosted on the first sub-cluster were used for performing the actual computation, while



(a) mpiBLAST



(b) ParaMEDIC

Figure 4.4: Breakup of Performance with Network Delay

4 processors on the second sub-cluster were used for the post-processing.

As shown in Figure 4.3, when the network delay between the two sub-clusters is low, mpiBLAST outperforms ParaMEDIC. This is expected since ParaMEDIC requires additional computation for converting the search results to metadata and converting the metadata back to the final output. However, as the network delay increases, ParaMEDIC outperforms mpiBLAST. In fact, for a network delay of 100ms, ParaMEDIC outperforms mpiBLAST by a factor of 2.26. This improvement is attributed to two factors. First, high network latency causes degradation in the filesystem communication and synchronization operations required when data needs to be written or read from the server. Second, the total amount of data written in mpiBLAST over the I/O subsystem is much higher as compared to ParaMEDIC, since ParaMEDIC only writes metadata which is significantly smaller than the final results to the filesystem.

To further understand these results, we show the performance breakdown of the time taken by mpiBLAST and ParaMEDIC in Figure 4.4. As shown in Figure 4.4(a), for mpiBLAST, as the network delay increases, the I/O time increases very quickly. Thus, though the computation time does not change much, the overall execution time suffers. On the other hand, for ParaMEDIC (Figure 4.4(b)), the computation time, the I/O time, and the post-processing time required to handle the metadata are nearly constant for all values of network delays. This is expected since the only component in ParaMEDIC that would be affected by the network latency is the post-processing, since it requires moving the metadata from the compute workers to the I/O workers. And because the metadata amount is very small (few KB), this time typically does not make any difference to either the post-processing time or the overall execution time of the application.

Trading Computation to I/O

We analyze the performance of mpiBLAST and Para-MEDIC by varying the ratio of computational resources allocated to the actual application processing vs. the resources allocated for post-processing. For all experiments in this section, for ParaMEDIC, we allocate four processes for performing the post-processing. The number of processes allocated for the actual application computation is varied from 16 to 80. That is, the ratio of resources allocated for actual application computation to post-processing is varied from 4:1 to 20:1. For mpiBLAST, on the other hand, all of the processes are allocated for the actual

application computation. That is, the native mpiBLAST implementation gets four extra processes for application computation as compared to ParaMEDIC.

The ratio of resources allocated to application and post-processing essentially determines the trade-off in computation time to I/O time. Specifically, a large ratio means that more resources are given for application processing, thus the resources given for metadata generation and management is minimal. This implies that the application execution time will be lesser, while the amount of data that needs to be moved over the distributed environment will be large. Similarly, a small ratio means that lesser resources are given for application processing, thus the resources given for metadata generation and management is high.

Figure 4.5 shows the performance of the two schemes as the ratio of resources alloted to application and post-processing is varied. As shown in the figure, when this ratio is low, mpiBLAST outperforms ParaMEDIC. However, as the ratio increases the performance of mpiBLAST degrades faster than ParaMEDIC, and it is eventually outperformed by ParaMEDIC.

This behavior is related to the available compute resources for execution. Specifically, when the total number of compute resources is *N*, ParaMEDIC uses (N-4) of them for application computation and 4 processes for metadata post-processing. Thus, when N is very large, the increase in computation time caused by using resources (approximately N / (N - 4)) is not very high. However, when N is small, the increase in computation time can be substantial. For example, when N is 8 processes, ParaMEDIC uses only 4 processes for the application processing while mpiBLAST uses 8. Thus, the computation time taken by ParaMEDIC would be nearly twice that of mpiBLAST. This overshadows any benefit in the I/O time ParaMEDIC can bring about, causing it to deliver worse performance than mpiBLAST. In summary, ParaMEDIC is most effective only when the number of resources used for application processing are sufficiently large as compared to the number of resources used for post-processing.

Impact of Output Data Size

In this section, we vary two parameters that affect the output data size: (i) number of input query sequences provided by the user and (ii) number of output query sequences requested by the user, and study their impact on the performance of mpiBLAST

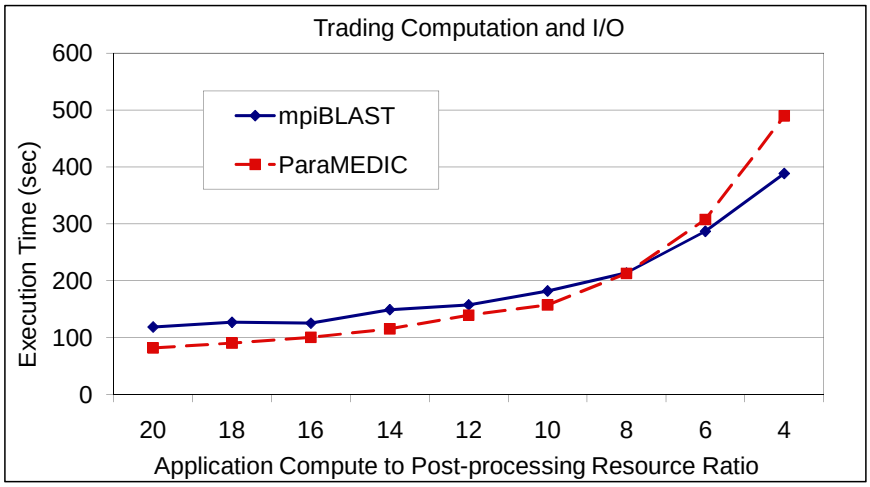


Figure 4.5: Varying the Number of Worker Processes

and ParaMEDIC. Varying the number of sequences in the input query increases the search time for both mpiBLAST and ParaMEDIC. However, it is also expected to impact the post-processing time for ParaMEDIC. Thus, increasing the query size is expected to affect the computation time of ParaMEDIC more than that of mpiBLAST. At the same time, an increase in the query size also typically results in more output. This, on the other hand, can potentially impact mpiBLAST more than ParaMEDIC. Figure 4.6(a) shows the performance of the two schemes with increasing number of input query sequences (depicted by input query size). We see that while the increase in the input query size increases the execution time of ParaMEDIC, it has a more drastic effect on mpiBLAST. Thus, as the query size increases, we notice that the performance difference between the two schemes increases, with ParaMEDIC outperforming mpiBLAST by about 66% for a 100KB query file size.

Figure 4.6(b) shows the impact of increasing the number of requested output result sequences. Increasing the number of output result sequences does not increase the computation too much, while it can increase the amount of I/O. Thus, because the I/O cost for ParaMEDIC is very low, increasing the number of output result sequences does not vary its performance too much. On the other hand, since the I/O cost for mpiBLAST is very

high, increasing the number of output result sequences significantly affects its performance.

Impact of Encrypted Filesystems

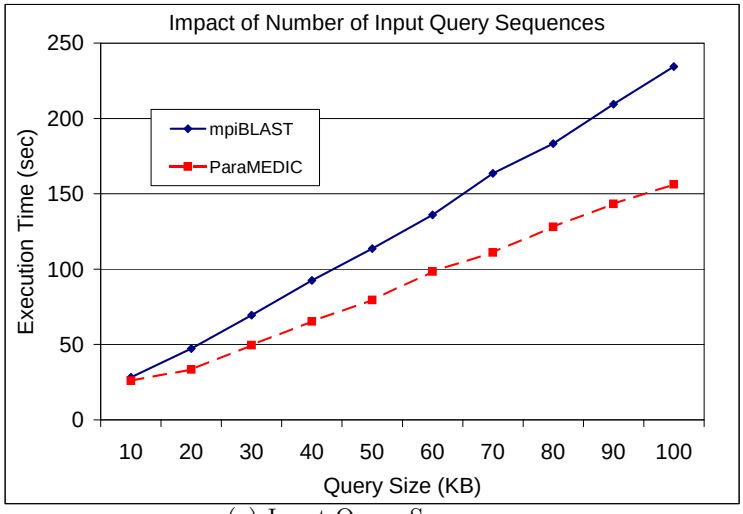
For distributed filesystems that span unsecure network connections (such as the Internet), using data encryption to protect transmitted data is a common occurrence in several environments such as government national laboratories and other secure facilities such as those demonstrated in §4.5.3.

Figure 4.7 shows the impact of such data encryption on the performance of the two schemes. As shown in the figure, the performance of the two schemes is similar to the case where there is no file encryption (except that the performance of mpiBLAST degrades faster). This is attributed to the data encryption overhead. That is, since all the data that is being transmitted has to be encrypted and the amount of data transmitted by mpiBLAST over the unsecure network is significantly larger than ParaMEDIC, encryption affects mpiBLAST more significantly as compared to ParaMEDIC.

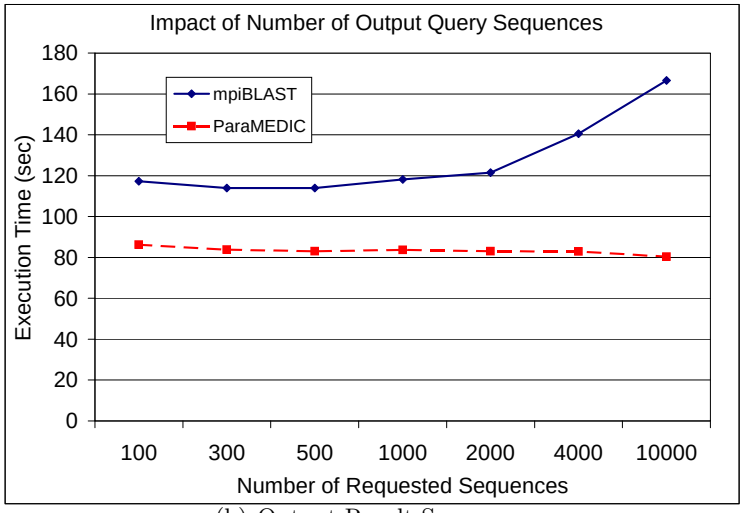
4.5.3 Distributed Setup from Argonne and VT

Here we evaluate mpiBLAST and ParaMEDIC on a distributed system between Argonne National Laboratory and Virginia Tech connected over Internet2. Since the network connecting the two clusters is *not* secure, data encryption is used to protect the data transmitted over this network.

As shown in Figure 4.8, ParaMEDIC significantly outperforms mpiBLAST in this environment. Further, as the query size increases, the performance difference between the two schemes increases. For a query size of 100KB, we observe more than a *25-fold improvement* in performance for Para-MEDIC as compared to mpiBLAST. This difference is attributed to multiple aspects. First, given that the network connection between the two sites is shared by other users, the effective network performance achievable is usually much lower than within the cluster. Thus, with mpiBLAST transferring the entire output result over this network, its performance would be heavily impacted by the network performance. Second, since data communicated is encrypted, mpiBLAST also has to pay the penalty for such encryption. Though ParaMEDIC also pays such data encryption penalty, the amount of data it transfers is significantly lesser, and hence the penalty is lesser as well. Third,



(a) Input Query Sequences



(b) Output Result Sequences

Figure 4.6: Varying the Number of Requested Sequences

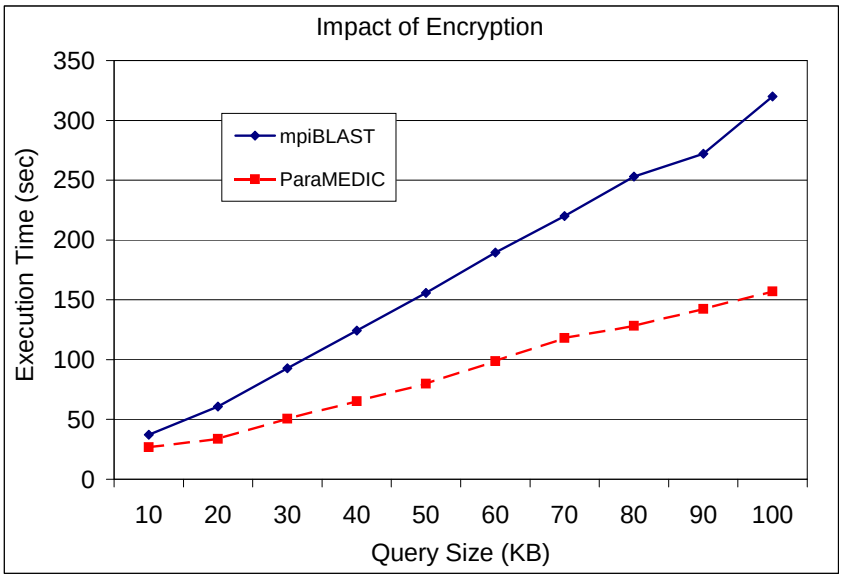


Figure 4.7: Impacted of Encrypted Filesystems

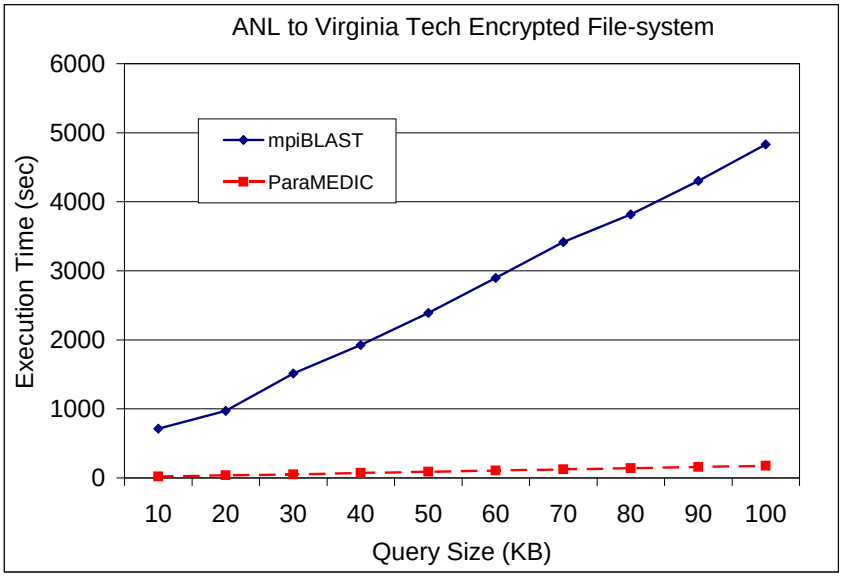


Figure 4.8: Argonne to Virginia Tech Encrypted Filesystem

the distance between the two sites causes the communication latency to be high. Thus, file-system communication and synchronization messages tend to take a long time to be exchanged resulting in further loss of performance.

4.5.4 TeraGrid Infrastructure

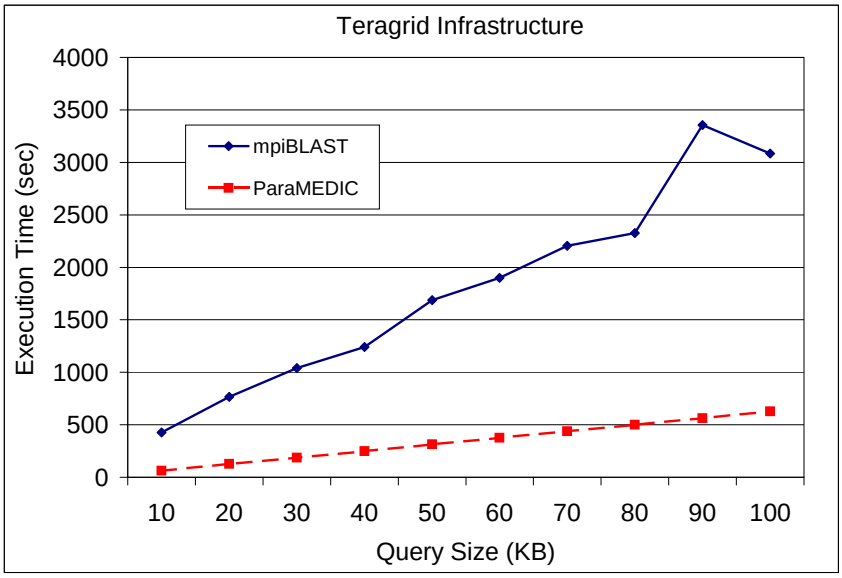
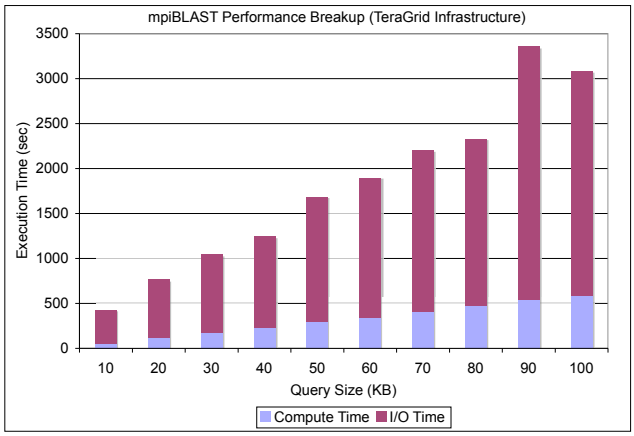


Figure 4.9: NSF TeraGrid using U. Chicago and SDSC

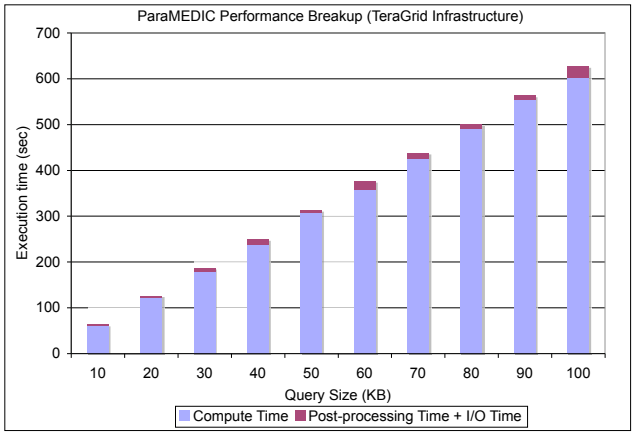
The TeraGrid infrastructure represents a widely used real environment for several compute- and I/O-intensive applications. As described in §4.2, a GPFS-based distributed filesystem is hosted at San Diego Supercomputing Center (SDSC), which can be accessed from all facilities, and forms a part of the TeraGrid facility. For the experiments in this section, we utilized the nodes at the University of Chicago and SDSC.

In this experiment, both mpiBLAST and ParaMEDIC perform their application computation on the University of Chicago nodes. However, mpiBLAST directly writes the output data to the global GPFS file-system. ParaMEDIC, on the other hand, converts the output data to metadata, transfers the metadata to SDSC, and re-converts the metadata to the final output at SDSC.

Figure 4.9 shows the performance of mpiBLAST and Para-MEDIC on TeraGrid.



(a) mpiBLAST



(b) ParaMEDIC

Figure 4.10: TeraGrid Infrastructure Performance Breakup

While the final output is written to the same global filesystem in both cases, mpiBLAST suffers because the application processing nodes at University of Chicago are performing the I/O for the output results. Since these nodes reside on a remote cluster as compared to the physical filesystem, their I/O performance is limited resulting in an overall degradation in execution time. For ParaMEDIC, on the other hand, since the post-processing nodes are performing the I/O for the output results, the amount of time taken is significantly smaller. For a query file size of 100KB, ParaMEDIC outperforms mpiBLAST by five-fold.

Figure 4.10 shows the performance breakdown of the two schemes. As the query size increases, the computation time for both mpiBLAST as well as ParaMEDIC increases. However, for mpiBLAST, the I/O time also increases very quickly. On the other hand, for ParaMEDIC, there is practically no difference in the I/O time with increasing query sizes. That is, ParaMEDIC is only minimally impacted by the limited I/O of the subsystem and it efficiently distributes its computational resources across the system to achieve high performance.

Chapter 5

Adaptive Request Scheduling for Clustered BLAST Web Services

5.1 Introduction

Our optimizations on massively parallel sequence search allow users to unleash the power of supercomputer in solving highly resource-demanding sequence-search jobs. In practice, life scientists also need to routinely search a small number of query sequences against public sequence databases in their daily research. Batch sequence searching on supercomputers is not the best way to address this need. First, those small jobs often play an important role in the sequence analysis work flow and thus require fast response time. For a batch sequence-search job, the response time is typically long because of the job submission and queuing overhead. Second, because of the high frequency of search, scientists would prefer handy access to the parallel computing power so that they can focus on their science research rather than learning and exercising the parallel job management.

An alternative approach to providing fast interactive sequence searching is to host parallel sequence-search web services on dedicated supercomputing resources (e.g. clusters). Through the ease-of-use web interface, the parallel processing power is available at scientist's fingertips. Unlike batch sequence searching, where data is typically not shared between jobs, sequence search web services have the opportunity to improve query response time by exploiting efficient sharing of the common databases across different requests. Finally, sequence-search web services can also enable efficient compute resource sharing and utilizing

between research institutions as well as facilitate collaborative sequence exploring efforts.

One of the most successful stories about the online sequence-search services is the NCBI online BLAST server [63, 64]. Built on top of a farm of hundreds of Linux workstations, the NCBI BLAST server is capable to process hundreds of thousands of queries submitted worldwide per day [3]. However, due to the high volume request traffic, the NCBI BLAST server suffers large variations in query response time, which has been verified with our query experiences. To obtain better service quality, research institutions have been moving to host their own sequence-search web servers on commodity clusters. Given a parallel sequence-search server shared by many users, efficient request scheduling is crucial to the service quality in terms of the average request response time. However, existing scheduling strategies from two related application fields, namely commercial clustered web servers and space-shared parallel computers, are inadequate for this new type of workload. Below we briefly describe the reasons (more detailed discussion will be given in Chapter 6).

Sequence-search web services are both computation- and data-intensive, performing non-trivial algorithms over large amounts of shared data. In contrast, commercial web servers typically stream contents or perform low-cost relational database queries. Hence their scheduling algorithms concentrate on data locality optimization and load balancing. Also, a back-end server node usually handles many client requests simultaneously with multiple open connections. With sequence-search services, the CPU and the memory resources required to timely process a request are often far beyond those can be offered by a single node. Consequently, a group of these nodes is dedicated to every request in a tightly synchronized manner.

In this sense, request processing in sequence-search web services is closer to batch job processing on parallel computers, however with two major differences. First, on general-purpose parallel computers, batch jobs are mutually independent and rarely share data. Second, as shared computation platforms, parallel computers have no knowledge regarding each job's computation and I/O requirements, and the resources requested by each job (such as the number of processors and the maximum run time) are specified explicitly in job scripts. Therefore batch job scheduling usually pays no attention to data locality issues and has no control over the level of concurrency in each job. With sequence-search services hosted by specialized data centers, data sharing is common and the parallel web server has much more knowledge about the services it provides.

Therefore, parallel sequence-search web services require careful examination of the intertwined computation and data management issues in making scheduling decisions. In this work, we extended scheduling algorithms to work for parallel scientific web services, from those designed for the commercial cluster web servers and batch processing parallel computers. By adopting a novel combination of these extended algorithms, a parallel scientific web server will take into consideration both the data and the computation aspects: data locality, parallel execution efficiency, and load balancing. In addition, the combined strategies work fully adaptively, automatically adjusting scheduling strategies according to the server load levels and dynamic data access patterns.

It worths noting that the request scheduling problem here is different than the one we investigated in Chapter 3 for massively parallel sequence search. The latter deals with a single batch job with a large amount of input query sequences and a single database, with a main goal to improve the overall system throughput by efficiently utilizing the computation and I/O resources in a supercomputer. The scheduling problem here deals with a large number of small sequence-search jobs from many users, each searching against an individual sequence database. The goal is to optimize the average request response time by exploiting data locality between jobs and parallel efficiency of sequence-search algorithms.

We implemented our proposed scheduling algorithms, along with baseline strategies to compare with, in a parallel BLAST server prototype. Our experiments on a cluster server performing parallel BLAST revealed that a careful choice in query concurrency and database-to-processor assignment may easily result in a dramatic difference in the average query response time. We confirmed that different query arrival rates and query composition ask for specialized strategies, and there are no "one-size-fits-all" solutions. The combination of the proposed adaptive strategies, however, achieves the best or close-to-best performance across a wide range of system load levels, with a several-fold improvement in average query response time in many cases.

5.2 Parallel BLAST Web Server Architecture

Figure 5.1 illustrates the parallel BLAST web server architecture targeted in our study, with sample query and partial output. As in a typical cluster setting, each node has its own memory and local disk storage, as well as access to a shared file system. One

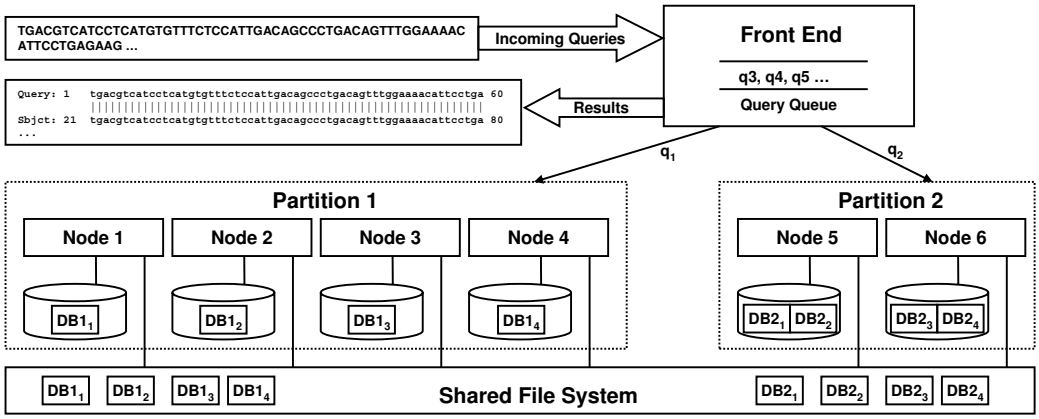


Figure 5.1: Target parallel BLAST web server architecture

of the cluster nodes serves as the front-end node, which accepts incoming query sequences submitted online, maintains a query waiting queue, schedules the queries, and returns the search results. The other nodes are back-end servers, often called "processors" in the rest of the paper for brevity.

For each query, the front-end node determines the number of processors to allocate, selects a subset of idle back-end nodes (called a *partition*) when they are available, and assigns these nodes to execute this query. After the parallel BLAST search, the results are merged by one of the nodes in the partition and returned to the client via the front-end node.

To save the database processing overhead, all the sequence databases supported by the parallel BLAST web server are pre-partitioned and stored in the shared storage. Figure 5.1 shows two sample databases, each partitioned into 4 fragments. The required database fragments will be copied to the appropriate back-end nodes' local disk before each query is processed, and are cached there using a cache management policy. Existing parallel BLAST implementations allow multiple database fragments to be "stitched" into a larger virtual fragment with little extra overhead. Therefore for the maximum flexibility in scheduling without creating physical fragments of many different sizes, we partition the database into the largest number of fragments allowed to be searched in parallel. To simplify the scheduling and to achieve better load balance, both the database fragmentation and processor allocation are based on power-of-two numbers, which is natural considering the way clusters are purchased or built. Note that the fragments combined into a larger virtual

fragment do not need to be in consecutive order. For example, when 16 processors are assigned to search a certain query against a database partitioned 64-way in a 64-processor cluster, one of them may be assigned to search fragments 0, 8, 45, and 57.

Assumptions: Before we move on to the scheduling strategies, we summarize assumptions made in this study: First, we assume a homogeneous environment, which is true for most clusters. Second, in this work we discuss the scenario where the entire collection of databases can be accommodated at each cluster node's local disks.[1] This is likely the case for parallel BLAST servers, as the total size of formatted NCBI sequence databases is currently around 100GBs, while a cluster node can easily have hundreds of GBs of local disk space today. Finally, to simplify query workload generation, we assume that each query contains only one sequence. Although existing BLAST web servers may allow users to upload multiple query sequences, the standard NCBI BLAST engine processes input queries sequentially. The difference in search time between the shared and separate BLAST sessions for multiple query sequences is not significant and mainly lies in the initialization overhead. Our research results can be easily extended to handle multiple-sequence requests. In the rest of the paper, we use the terms "request" and "query" interchangeably.

5.3 Scheduling Strategies

In this section, we present scheduling strategies for parallel scientific web services, using parallel BLAST server as a case study. We extend two existing scheduling algorithms and integrate them to design adaptive algorithms that automatically adjust to various query workloads and cluster configurations. Like in many existing request scheduling studies, our major goal is to optimize the average query response time.

In Section 5.3.1 and Section 5.3.2, we discuss our extended scheduling algorithms respectively. The first one comes from the commercial cluster web server community and performs *data-oriented scheduling*. It determines which processors should be allocated for a specific query, considering existing data cached at these processors and their current load. The second one comes from the space-sharing parallel job scheduling community and performs *efficiency-oriented scheduling*. It determines the desired level of concurrency for

[1] For systems equipped with insufficient local storage, we have developed additional optimizations, as described in our technical report [15].

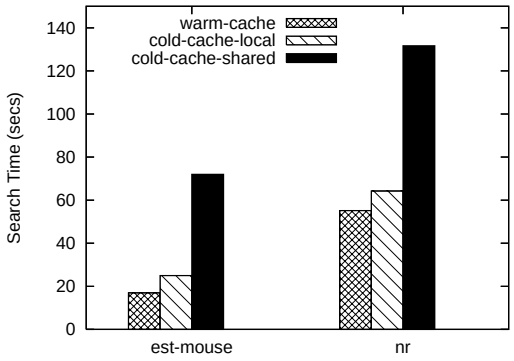


Figure 5.2: Impact of data placement on the BLAST performance.

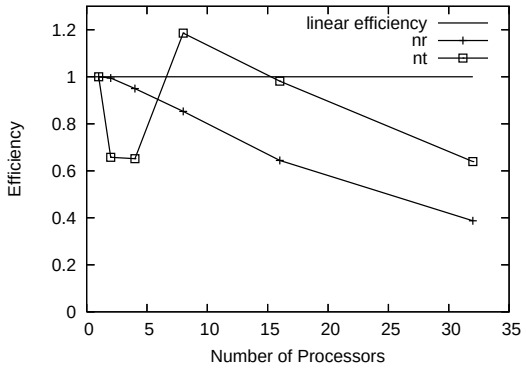


Figure 5.3: Parallel execution efficiency of BLAST

processing a query, considering the specific query workload and the current system load. Both algorithms are extended substantially to fit the scenario of parallel scientific web services. Then in Section 5.3.3, we discuss our overall scheduling scheme and describe how we integrate the two scheduling algorithms.

5.3.1 Data-Oriented Scheduling

Like in other distributed or cluster web servers, data locality is a key performance issue in parallel BLAST web servers. Figure 5.2 demonstrates the impact of going down the storage hierarchy: main memory, local file system, and shared file system. The experiments use sequential NCBI BLAST to search the `est-mouse` and `nr` databases, which can fit into the memory of a single processor. For each case, 10 sequences randomly sampled from the database itself are used as queries, and the average search time is reported. In the "warm-cache" tests, we warm up the file system buffer cache with the same query before taking measurements, and in the "cold-cache" tests we flush the cache first. For "cold-cache-shared", we force loading the database from the shared file system. The results indicate that improving file caching performance and in particular, reducing remote disk accesses can significantly improve the search performance.

As mentioned earlier, in this paper we focus on the scenario where the entire set of databases hosted by a parallel scientific web server can fit into the per-node local disk space. Still, only a small fraction of those databases can be buffer-cached in the main memory, and scheduling must be performed considering the data locality issue. One intuitive locality-aware optimization is to assign queries targeting different databases to disjoint pools of processors and let each processor pool search the same database repeatedly. This way, the

effective working set of each processor is reduced. Creating static per-database processor pools, however, is not flexible enough to handle the dynamic online query composition and will likely cause serious system underutilization.

A similar problem has been addressed regarding general-purpose content-serving cluster web servers. In this paper, we extend the LARD algorithm for content request distribution proposed by Pai et al. [65] to the parallel scientific web service context. Given a set of back-end servers, the LARD algorithm assigns partitions of hosted targets to subsets of these servers. An incoming web request will be routed to one of the servers assigned to its target, or the least loaded server if it is the first request of the given target. Load balancing is performed periodically to move requests from heavily loaded servers to lightly loaded ones. LARD exploits data locality to improve the server performance by assigning requests of the same target to the same set of processors.

Two major differences make our target system considerably more complex than a general-purpose cluster web server. First, multiple processors need to be co-scheduled to queries or co-transferred between pools. Second, a processor can handle only one query at any given time. Therefore queries cannot be piled to server nodes as they arrive, but need to wait for dispatch.

To handle these requirements, we extend LARD to a new algorithm called PLARD (Parallel LARD). To perform locality-aware assignment and load balancing, PLARD adopts a two-level scheduling mechanism. It establishes one global query queue (global_queue) and multiple per-database query queues (queue[DB_i]). Queries will be first appended to the global queue, and subsequently dispatched to one of the per-database queues. Similarly, because servers need to be assigned in groups, PLARD manages a global idle processor pool (global_pool), and multiple per-database processor pools (pool[DB_i]). Initially, all the processors are in the global pool. A scheduling operation will be triggered by either a query arrival or a query completion. Algorithm 2 gives the detail of the process of scheduling one query from the global queue.

Queries in the global queue will be scheduled in the first-come-first-serve (FCFS) order. When there are not enough resources for the next query, the scheduling attempt is aborted and the global scheduler waits until a query completes. This helps ensure fairness and prevents starvation. Also, this allows the recommended partition size to be recalculated as the system load changes.

Before moving a query from the global queue to a per-database pool, a recommended partition size will be calculated by the function get_recommended_size(). This function determines how many processors should be allocated to a target database, using algorithms such as the ones described in the next section. The target database-pool will be enlarged if the pool size is less than the recommended size. In case there are not enough processors to allocate from the global pool, the algorithm will seize processors from the most lightly loaded pool if there are fewer queries waiting in that pool's local queue than those waiting for the target database in the global queue.

Algorithm 2 PLARD

```
fetch the next query q from global_queue
partition_size ← get_recommended_size()
m ← the number of queries waiting for q.target_db in global_queue
candidate_queues ← ⋃ queue[DB_i], where DB_i not equal to q.target_db
increase_size ← partition_size - pool[q.target_db].size
if increase_size > 0 then
   while global_pool.size < increase_size and candidate_queues not empty do
      size_needed ← increase_size - global_pool.size
      find queue[DB_j] ∈ candidate_queues with smallest queue length
      if m > queue[DB_j].length then
         num_idle ← the number of idle nodes in pool[DB_j]
         S ← release_idle_nodes(DB_j, min(num_idle, size_needed))
         add S to global_pool
      end if
      remove queue[DB_j] from candidate_queues
   end while
   if increase_size ≤ global_pool.size then
      A ← allocate increase_size processors from global_pool
      add A to pool[q.target_db]
   end if
end if
if pool[q.target_db] is not empty then
   append q to queue[q.target_db]
end if
balance_load()
```

After a query is assigned to a per-database processor pool, it goes to the local queue of that pool and is scheduled using an internal scheduling algorithm (such as a fixed partitioning policy or RMAP, as presented in the next section). This way, a relatively stable subset of server nodes are assigned to work on a certain database, maximizing the use of their collective buffer cache space.

Like in the original LARD, every time a query is scheduled the system per-

forms load balancing. In PLARD, we move processors from the most lightly loaded pool (pool[DB_{min}]) to the most heavily loaded pool (pool[DB_{max}]), if one of the following conditions is satisfied:

1. queue[DB_{max}].length - queue[DB_{min}].length > T and
   queue[DB_{max}].length $\geq 2\times$ queue[DB_{min}].length, or

2. queue[DB_{min}].length = 0 and queue[DB_{max}].length > 1

T in the above is a configurable threshold, which is set as 10 in our implementation. The number of processors moved during load balancing is set to be P_{min} of DB_{max}, where P_{min} is the minimum partition size allowed for a given database as described in Section 5.3.2. This helps reduce the internal fragmentation of a database pool during load balancing.

5.3.2 Efficiency-Oriented Scheduling

PLARD helps us optimize query processing performance by maximizing the use of cached data and improving load balance between server nodes. However, it does not consider the parallel processing scalability of the scientific applications that service the web requests. The latter turns out to be crucial in deciding how many processors should be allocated to each individual query, and can have a significant impact on the parallel web server's performance.

We illustrate the argument by examining parallel BLAST's performance scalability. Like most parallel applications, it is subject to the performance tradeoff between absolute performance and system efficiency when the level of concurrency increases. One obvious explanation is the higher parallel execution overhead associated with searching a single query using more processors. In addition, as BLAST performs top-k search, the task of processing and filtering of intermediate results grows with the number of processors. Figure 5.3 illustrates the performance trend of parallel BLAST from searching two widely used databases, the NCBI `nr` and `nt`, as benchmarked on our test cluster (to be described in Section 5.4.1). For each search workload, we plot the *efficiency*, which is defined as parallel speedup divided by the number of processors. Therefore a perfect linear efficiency is a flat line. For both `nr` and `nt`, the efficiency slides steadily as more processors are used for each query.

Systems such as the NCBI BLAST server reported periodic variances in the query arrival rate [63]. One intuitive heuristic is to control the number of processors allocated to each query based on the current system load: when the load is light, allocate more processors for smaller query response time; when the load is heavy and queries are piling up in the queue, allocate fewer processors for better system throughput (and consequently better average response time). This intuition is backed up by queuing theory and has been adopted in adaptive partitioning algorithms for parallel job scheduling [66]. In this work, we select the MAP algorithm [67], which improves upon the above work, as our base algorithm.

With MAP, both the waiting jobs and the jobs currently running are considered in determining the system load. It chooses large partitions when the load is light and small ones otherwise. More specifically, for each parallel job to be scheduled, a target partition size is calculated as

$$target\_size = Max(1, \lceil \frac{n}{q + 1 + f * s} \rceil),$$

where n is the total number of processors, q is the waiting job queue length, s is the number of jobs currently running in the system, and f $(0 \leq f \leq 1)$ is an adjustable parameter that controls the relative weight of q and s. In our experiments, we set the f value as 0.75, as recommended in the original MAP paper [67]. Once the target partition size is selected, the front-end node waits until these many processors become available to dispatch the query.

One may notice that in Figure 5.3 the `nt` curve does not monotonically decrease. Instead it peaks at 8 processors, with a super-linear speedup at that point. This is due to that the `nt` database cannot fit into the aggregate memory of 4 or fewer processors on our test platform. As BLAST makes multiple scans and random accesses to the sequence database, out-of-core processing causes disk thrashing and significantly limits the search performance. The `nr` database is much smaller and can be accommodated in a single compute node's memory, therefore does not show the same behavior.

This motivates us to propose Restricted MAP (RMAP), which augments the base MAP algorithm with a database-dependent and machine-dependent memory constraint. For a given database supported by a given cluster server, we select P_{min} and P_{max}, which define the range of partition sizes (in terms of the number of processors) allowed to schedule queries against this database. P_{min} is the smallest number of processors whose aggregate memory is large enough to hold the database. P_{max} is determined by looking up the saturation

point in the speedup chart: it is the largest number of processors before the absolute search performance declines. In other words, after this point deploying more processors will not produce any performance gain. An initial benchmarking is needed to set P_{max} for each database, which is feasible considering the total number of different databases supported by a web server is often moderate[2].

For each query scheduled, when there are more idle processors available than p, the desired partition size calculated, RMAP adopts a simple node selection strategy called FA (First Available), where the first p idle processors by the processor rank will be assigned to work on the query. Database fragments will be assigned to these processors in a round-robin manner.

5.3.3 Combining PLARD and RMAP

We integrated PLARD and RMAP in our two-level query scheduler implementation for the parallel BLAST server prototype.

As shown in Algorithm 2, when dispatching a query from the global queue to a particular DB queue, the RMAP algorithm is first used to calculate a recommended partition size based on the global system state. More specifically, the queue length(q) is calculated by summing up all queries in the global queue and local DB queues, and the number of queries in the system(s) is the sum of queries being searched at all DB pools. If the number of idle processors in the processor pool of the target DB is smaller than the recommend partition size, the scheduling algorithm seeks to assign more processors to this pool by acquiring idle processors from the system idle processor pool and/or other relatively lightly-loaded DB pools.

When a partition with the recommended size can be provided, the query is moved into the local DB queue. There the RMAP algorithm will be called again to determine a proper partition size in local scheduling. At this point, each local RMAP scheduler uses the local system state, namely the local DB queue length as q and the number of queries being serviced in the local processor pool as s.

With this two-level scheduling approach, we adapt simultaneously to the intensiveness and the database access pattern of the dynamic query workload by leveraging strengths

[2]The number of all sequence databases offered by the NCBI web search is 21 at the time this paper is written.

of both RMAP and PLARD. The two algorithms complement each other nicely under the new scheduling framework.

5.4 Performance Results

5.4.1 Experiment Configuration

Table 5.1: Database characteristics. Note the P_{min} values are multiples of 2, this is because our experiments are performed on a two-way SMP cluster, and we found using a compute node (2 processors) as the smallest scheduling unit yields better performance than does using an individual processor, as the former choice has better data locality.

Name	Type	Raw Size	Formatted Size	P_{min}	P_{max}
env_nr	P	1.7GB	2.5GB	2	32
nr	P	2.6GB	3.0GB	4	32
est_mouse	N	2.8GB	2.0GB	2	16
nt	N	21GB	6.5GB	8	32
gss	N	16GB	9.1GB	8	32

In our experiments, we use five genomic sequence databases downloaded from the NCBI public sequence repository. Table 5.1 summarizes several basic attributes of these databases. Among them, the first two are protein sequence databases (type "P") and the other three are nucleotide sequence databases (type "N"). The two types of the databases are searched using the `blastp` and `blastn` algorithms respectively. The size of each database shrinks after the database is formatted for search using the standard `formatdb` tool. For each of the databases, we also give the P_{min} and P_{max} pair, which defines the processor partition size range. As discussed in Section 5.3.2, P_{min} is determined by the memory constraint and P_{max} is determined by benchmarking the parallel execution scalability of the individual database search workload.

The parallel BLAST software we used is the popular mpiBLAST tool [55, 8], available at http://mpiblast.org/. For queries, we sampled 1000 unique sequences from the five databases, with the number of samples from each database proportional to the formatted database size. Since sequence databases are constantly appended with newly discovered sequences, we hope this sampling method resembles the composition of real BLAST search workloads, which are driven by sequence discoveries. We compose online query traces by drawing queries randomly from this pool of unique sequences, setting the

arrival interval with the Poisson distribution.

To create traces with the desired arrival rates, we benchmark the maximum throughput of the whole system. This maximum throughput is calculated in an aggressive manner: we measure the maximum throughput of each database' search workload by executing the corresponding subset from the 1000-query pool on the whole cluster using the smallest partition size (P_{min}). This way the system achieves best efficiency and data locality with the single-database workload and small partition size. We then derive the multi-database maximum throughput by taking a weighted average of the single-database peak throughput, according to the number of queries going to each database.

Unless noted otherwise, the experiments are performed using query traces that contain 600 query sequences sampled from the 1000-query pool above. Note that many of the charts use log2 scale on the y axis, due to the large distribution of performance numbers under different system load levels.

5.4.2 Test Platform

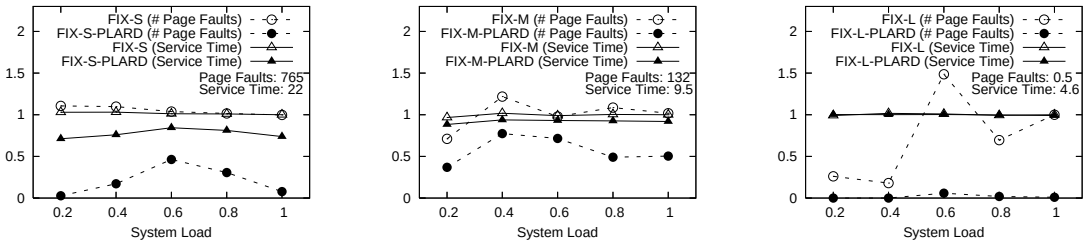
Our experiments were performed on the Orbitty Linux cluster located at North Carolina State University. Orbitty consists of 20 compute nodes, each equipped with dual Intel Xeon 2.40GHz processors sharing 2GB of memory. Due to its target workload, this cluster has 400GB per-node local storage space, which is large enough to host the entire collection of NCBI sequence databases. The interconnection is Gigabit Ethernet and a shared storage space of over 10TB is accessed through a Lustre server.

5.4.3 Data-Oriented Scheduling Results

First, we examine the effectiveness of improving data locality in query processing, by showing the impact of PLARD on three versions of fixed partitioning strategies. With fixed partitioning, the number of processors allocated to queries against the same database is fixed throughout the run. For each database, we choose three fixed partition sizes within its partition size range $[P_{min}, P_{max}]$: small (FIX-S), medium (FIX-M), and large (FIX-L).

Experiments are carried out using different levels of system load by adjusting the query arrival rate. A system load of 1 means the query arrival rate is equal to the maximum query throughput. All these strategies also use the default FA policy in selecting

idle processors to schedule.



(a) Small partition size (b) Medium partition size (c) Large partition size

Figure 5.4: Normalized average number of page faults and normalized average service time.

Figure 5.4 portraits the impact of PLARD on the fixed partition size algorithms' file caching performance. Since BLAST uses memory mapped files, the number of page faults is a good indication of the amount of file I/O performed to retrieve the database fragments. For each of the fixed algorithms, we plot the average number of page faults per node (dashed lines) and the average query service time (solid lines), with and without PLARD. All the page fault numbers are normalized against the page fault number of the original algorithm (without PLARD) with the system load of 1. The same applies to the service times. The absolute values of these two pivot numbers are marked in the charts.

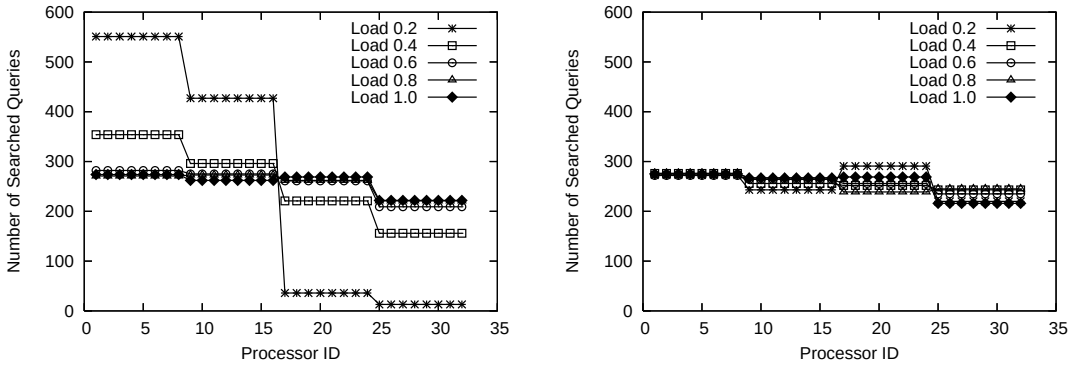
As expected, the PLARD algorithm does have a significant impact on the number of page faults. In particular, for FIX-S, the original page fault numbers of over 750 are reduced at least by half, and almost eliminated with the lightest and heaviest system loads. On average, the number of page faults is reduced by 79.87%. The original FIX-S page fault slightly declines as the system load intensifies since more processors will be actively used, and the chance of having cache hits increases due to the enlarged aggregate memory size, although there is no intentional, locality-aware query placement. With PLARD, however, the peak of page fault numbers appear in the medium load (0.6), where with the small partition size, the per-database processor pools are the most dynamic: processors are shifted between pools relatively more frequently, reducing the chances of cache hits within each database pool.

With FIX-M, the page fault reducing of PLARD is smaller but still considerable, with an average of 43.37% decrease. Here the peaks of the page fault numbers, both with and without PLARD, are different from those with FIX-S due to the larger partition sizes.

For example, the lightest load achieves the best data locality since the query load is rather concentrated on a group of processors, facilitating in-memory data reuse, while the size of the group is large enough to spread the databases out and reduce the data access working set per node.

With FIX-L, the databases are so spread out so that all the fragments needed by a processor are almost always in the memory. Although the normalized curves look dramatic, the absolute numbers are very small. Even without PLARD, the cache misses are negligible, with an average page fault count of 0.37.

The improvement of service times using PLARD is a direct result of the improved data locality, as PLARD does not affect the computation efficiency of each query's processing with the fixed-partitioning algorithms. The degree of the improvement, however, declines as the partition size selected increases. This is because the number of page faults goes down faster than the service time does when larger partitions are used. Therefore the impact of page fault reduction plays a smaller and smaller role.



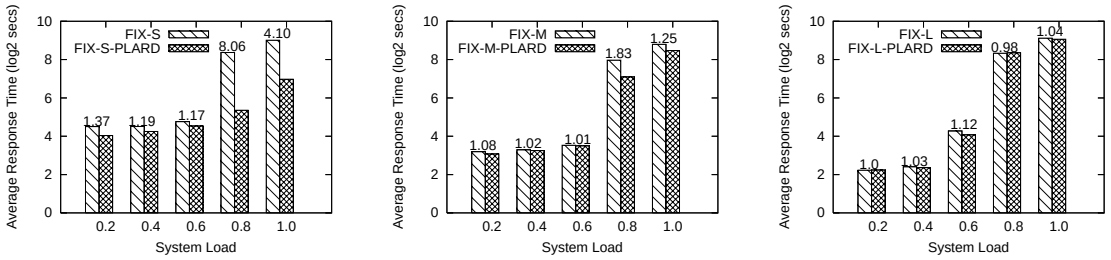
(a) FIX-M (b) FIX-M-PLARD

Figure 5.5: Query load distribution among processors with the medium partition size.

Figure 5.5 provides additional information about the effect of PLARD, from its load balancing aspect. We illustrate this using FIX-M, the algorithm using the medium partition size. As discussed above, with the FA policy for processor assignment, the query processing workload distribution is skewed at the load level of 0.2. Most queries are assigned to the first 16 processors, with an additional 100+ queries assigned to the first 8 (please recall

that the "medium partition size" varies from database to database). Heavier system loads force the queries to become more evenly distributed. With PLARD, the query processing assignments are well balanced among processors for all system load levels.

Now we take a look at the overall impact of PLARD, by comparing the average query response time before and after. Because long waiting time with heavy system loads caused a wide distribution of response time, we show the numbers in log scale, with the speedup factor brought by PLARD labeled at the top of each pair of bars.



(a) Small partition size (b) Medium partition size (c) Large partition size

Figure 5.6: Impact of PLARD on the average query response time. Note that the y axis uses the log2 scale, and the speedup factor brought by PLARD is shown at the top of each pair of bars.

Figure 5.6 shows the comparison, again for each of the FIX algorithms using multiple system loads. As expected, the largest improvements are found with FIX-S, where the average response time is reduced by up to 4.1 times. As we have seen from Figure 5.4, the largest enhancement to data locality and the average query service time occurs with the small partition size. The changes in service time, in turn, has a varying impact on the query response time. With heavier loads, the reduced service time has a rather dramatic effect on decreasing the queue length and average query wait time. With light loads, the enhanced service time does not affect the per-query wait time much. For FIX-M, the best improvement is observed at the load of 0.8, with a speedup factor of 1.83. Not surprisingly, PLARD does not bring significant improvement to FIX-L.

5.4.4 Efficiency-Oriented Scheduling Results

Now we examine the impact of RMAP by enabling PLARD for all tests and compare the three FIX algorithms with RMAP.

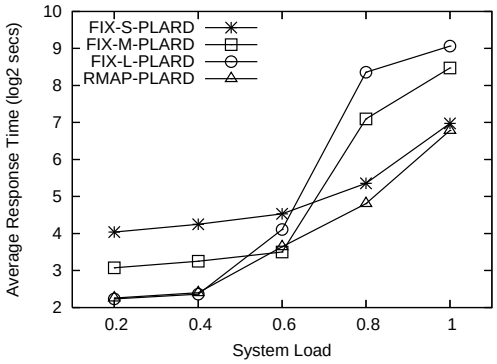


Figure 5.7: Performance of combined RMAP and PLARD with fixed arrival rates (y axis uses log2 scale.

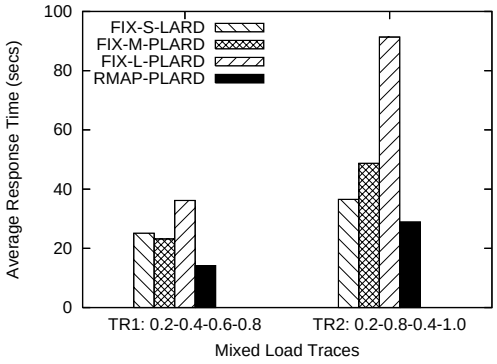


Figure 5.8: Performance of combining RMAP and PLARD on two 800-sequence traces with mixed arrival rates.

Figure 5.7 portraits the results. As expected, no single fixed partitioning strategy performs consistently well. When the system load is light, the large partition size works best by using a large number of processors to reduce each query's response time. As the load increases, first the medium, then the small partition size becomes the winner. With heavier loads, smaller partition sizes help achieving better overall resource utilization by improving the parallel execution efficiency. The performance difference is significant: across the x axis, the difference between the best and worst average response time among the fixed partitioning strategies varies between 3.5 and 8 times. RMAP, on the other hand, closely matches the best performance from the three fixed partitioning strategies by automatically adapting to the system load.

The only point where the RMAP performance is visibly lower than the best fixed partition size algorithm is with the medium system load (0.6). Because the trace we used is not even-paced, the medium load is an unstable case for RMAP, where the scheduler adjusts the partition size (in both directions) most frequently. With frequent partition size changes, cache contents cannot be well utilized and more cold misses are introduced.

To verify this, we take a closer look at the behavior of FIX-M-PLARD and RAMP-PLARD. Table 5.2 summarizes a group of measurements taken from the experiments using the two algorithms, at the system load level 0.6 and 0.8. Because the partition sizes are power-of-two numbers, we calculate the average partition size by taking the arithmetic average after performing the log2 operation. The "total service time" is calculated as the

Table 5.2: FIX-M-PLARD and RMAP-PLARD statistics at system load 0.6 and 0.8.

System Load	**0.6**		**0.8**	
Policy	FIX-M-PLARD	RMAP-PLARD	FIX-M-PLARD	RMAP-PLARD
Average # Page Faults	93.00	132.74	63.64	224.88
Average Service Time (s)	8.81	9.53	8.76	15.47
Average Waiting Time (s)	2.50	2.95	128.25	12.51
Average Response Time (s)	11.31	12.48	137.02	27.98
Total Service Time (s)	69936	78700	69657	63977
Average Partition Size (log2)	3.71	3.88	3.71	2.78

total computation resource usage in an experiment. For each query, we calculate its resource usage as the product of its service time and the number of processors it used. We sum up the resource usage of all queries in a trace as the total service time.

From the page fault counts, we see that RMAP does hurt the data locality at load level 0.6. Consequently, RMAP adopts a slightly larger partition size than FIX-M does, but has a 8% higher average service time. The service time increase causes a similar increase in the waiting time and average response time.

Interestingly, RMAP caused a much larger increase in the number of page faults at the load of 0.8, yet the average response time of RMAP is 5 times better than that of FIX-M. This is caused by that RMAP has better parallel computation efficiency there, which can be seen from the total service time: RMAP increased the total service time at 0.6 and decreased it at 0.8. Although the individual query's service time is longer than FIX-M, RMAP increases the whole-system throughput by automatically adopting a considerably smaller average partition size. With such a heavy system load, this had a dramatic effect on shortening the average query waiting time, and the average query response time consequently.

Finally, we evaluate the overall adaptivity of the combined RMAP-PLARD algorithm. Figure 5.8 shows two sets of experiments, each using a mixed load level trace containing 800 queries. In each trace, the average load level is adjusted several times, *e.g.*, from 0.2 to 0.4, 0.6, and finally 0.8, for four equal-length intervals (in terms of the number of queries). Trace 1 adopts such an monotonically rising system load as in the above example, while trace 2 has a repeated up-down pattern. Again, for such mixed load traces, none of the fixed partition size algorithms consistently win, and each of them may suffer trace intervals where the selected partition size is undesirable. RMAP, on the other hand, successfully adapts to the varying query intensiveness and significantly outperforms all the

fixed partition size algorithms, bringing an improvement factor of 1.63 and 1.26 in average response time over the best performing fixed algorithm for trace 1 and 2, respectively.

Chapter 6

Related Work

6.1 Genomic Sequence-Search Parallelization

As sequence database-search is computationally intensive, many parallel approaches have been investigated to cope with the rapid growth of sequence databases. Hardware-based solutions [68, 69, 70] parallelize the computation of comparing a single query sequence to a single database sequence. These solutions are highly efficient but require custom hardware such as Field-Programmable Gate Array (FPGA). The optimization techniques presented in Chapter 3 focus on software-based parallel solutions. Nonetheless, these techniques can be generalized to efficiently glue many hardware processing units together to provide even higher search throughput.

Early parallel sequence-search software adopted the *query segmentation* approach [71, 72, 73]. With this approach, each compute node is assigned the entire sequence database and a subset of query sequence. As the search computation of individual query sequences is completely independent, such an *embarrassingly parallel* approach can achieve very high parallel efficiency when the sequence database is relatively small. However, for sequence databases that are larger than the main memory of a computer node, query segmentation will incur high I/O overhead caused by the repeated scanning of the database when searching multiple query sequences. In addition, on platforms without virtual memory support, such as the IBM Blue Gene supercomputers, query segmentation is not applicable when the node memory is not sufficient to host the sequence database as well as the search intermediate results.

The limitations of query segmentation motivated the *database segmentation* [74, 55, 75, 8] approach. With database segmentation, the sequence database is segmented and distributed across computer nodes. Each node searches all query sequences against its own portion of the sequence database. By fitting large databases into the aggregate memory of multiple nodes, database segmentation eliminates the paging issue and allows timely sequence analysis to keep up with fast growing database sizes. However, this approach introduces computation dependency between individual nodes because the distributed results generated at different nodes need to be merged to produce final output. The parallel overhead caused by results merging will increase as the system size grows, consequently limiting the program's scalability on large-scale deployments. In Section 3.3.3, we introduced a light-weighted results merging design that can significantly improve the efficiency of merging a large amount of results data.

Recent efforts in designing large-scale sequence-search applications achieved high scalability by adopting a combination of both segmentation approaches. Rangwala et. al. developed a parallel BLAST implementation optimized for IBM Blue Gene/L [56]. In this work, all processors in the system are organized into different equal-sized groups. Each group searches a subset of query sequences against the sequence database with the efficient database segmentation approach introduced in pioBLAST [8]. This work adopts a static load balancing approach, where the query sequences assigned to each group will have approximately the same total length. To improve the data input performance on large system scales, a group of processors are dedicated as the I/O group. Processors in the I/O group first read the fragmented sequence database into their memory in parallel, then broadcast their own portion of sequence data to the corresponding processors in all other groups. All the searched results are buffered in processors' memory and written to the file system at the end with collective MPI I/O functions.

Oehmen et. al. reported ScalaBLAST [76], a highly efficient parallel BLAST built on top of the Global Array [77] toolkit. ScalaBLAST follows a similar design to combine the query segmentation and database segmentation approaches. However, instead of replicating the sequence database to all processor groups, ScalaBLAST have all processors share a single copy of the sequence database stored in a global array. ScalaBLAST adopts a slightly different static load-balancing approach, where each group is assigned a subset of query sequences containing the same amount of "work units". The work units of a query batch is

calculated based on a "*trial-and-error*" approach. Specifically, the work units of a batch of query sequences are calculated as the sum of the number of characters in all sequences, with an addition value of 225 per sequence. In ScalaBLAST, each processor will switch to output processing after searching every 20 query sequences. During the output processing, each processor in a group fetches the intermediate alignment results from the other processors in the group and converts them into the final output. ScalaBLAST maintains an individual output file for each process.

The above two massively parallel sequence search-tools both adopt static load-balancing approaches, assuming the execution time of a sequence-search task is predictable from the total sizes and/or the numbers of the input queries. Such a static load-balancing design scales well by avoiding scheduling dependency between different processor groups. However, our recent study discovered that for certain types of DNA sequence matching, there is no clear correlation between the mount of input data and the execution time of a sequence-search task [38]. In addition, tasks processing a same amount of input can have execution time differing by orders of magnitude. In fact, because of the heuristic nature of popular sequence alignment algorithms, we argue that it is hard to know a priori the execution time of a sequence search task. Consequently, mpiBLAST-PIO adopts dynamic load-balancing approaches and focuses on effectively reducing the associated scheduling overhead.

6.2 Noncontiguous I/O Optimizations

In many parallel scientific applications, processes need to access data files in a non-contiguous manner [78, 79, 80, 81, 82]. People have been investigating various optimizations to noncontiguous I/O accesses in both user level libraries and parallel file systems.

6.2.1 User Level Optimizations

There are two techniques widely used to optimize noncontiguous I/O performance used in popular parallel I/O libraries such as ROMIO [50]: *data sieving* and *collective I/O*.

Data Sieving

Data sieving was introduced in the PASSION I/O library [83]. It targets noncontiguous I/O requests issued from one process. Data sieving groups small noncontiguous I/O requests into large ones with the cost of redundant I/O. By doing so, it reduces the number of I/O requests sent to the file system and consequently improves the I/O performance.

For a read operation, data sieving fetches a large chunk of data that covers several pieces of closely located noncontiguous data and put it into a memory buffer. The actual needed data is then supplied to the applications from the memory buffer. Data sieving handles a write operation with a *read-modify-write* approach: 1) a large chunk of data that covers the range of multiple noncontiguous I/O requests is read into a memory buffer, 2) the application only modifies corresponding write regions in the data sieving buffer, and 3) the whole trunk of data in the data sieving buffer is written back to the file system. To avoid data corruption caused by concurrent accesses, the file region manipulated by data sieving needs to be locked during the read-modify-write procedure.

Data sieving aims at trading extra I/O data for large, sequential I/O requests. It works well when the noncontiguous requests issued in an I/O operation are relatively dense. However, when the noncontiguous requests are sparse, the cost of redundant I/O can surpass the benefit of requests aggregation and result in unsatisfactory I/O performance. In addition, for write operations, data sieving could suffer lock contentions when the I/O requests from multiple processes are highly interleaved. Data sieving is not suitable for parallel sequence-search applications because of their irregular I/O data distributions.

Collective I/O

Collective I/O was designed to improve parallel noncontiguous I/O performance. It takes advantages of an application's collaborative I/O access information to aggregate small I/O requests into large ones. Collective I/O is typically implemented with two-phase I/O [49] in parallel I/O libraries, but it can also be implemented at the disk level [84] or the server level [85].

Two-phase I/O services I/O requests from multiple processes with a communication phase and an actual I/O phase. For a collective read, the portion of the file that will be accessed by the collaborative requests is first split evenly into different *file domains*

and assigned to the involved processes. Each process first reads its assigned file domain, which is a large, contiguous chunk of data, into a memory buffer. In the second phase, each process sends the data in its file domain to other processes requesting for the corresponding data. For a collective write, the communication phase happens first. The involved processes perform in-memory data exchange so that each process will store a large block of write data in a memory buffer. The buffered data will be written to the file system in the I/O phase. The I/O data buffered on a process can be noncontiguous as well, in which case data sieving will be used in the I/O phase in parallel I/O libraries such as ROMIO [50].

Two-phase I/O can effectively improve I/O performance by aggregating small, noncontiguous in to large, contiguous ones. However, the I/O data needs to be transfered twice over the network. This extra network communication cost is usually worthy given that the memory access and the network transfer are much faster than the random disk I/O. In particular, the network communication is highly optimized on modern supercomputers.

The disadvantage of collective I/O is that it incurs implicit synchronization between processes. This is not an issue for applications with inherent synchronization in their compute kernels. As discussed in Section 3.3.4, the synchronization will incur considerable overhead in parallel sequence search as the search time of a query can be highly imbalanced between processes. One could argue that split collective I/O defined in the MPI-IO standard does allow applications to overlap computation with collective I/O operations without forcing synchronizations. The split collective I/O, however, is not available in popular parallel I/O libraries. In addition, the output data distribution varies from query to query in parallel sequence search, thus involved processes need to construct file views for every collective write operation. The function call (i.e. MPI_File_set_view) to construct the file view is synchronized by definition in the MPI standard.

6.2.2 File System Level Optimizations

There have been studies in optimizing noncontiguous I/O at the parallel file system level. Ching et. al. proposed a technique called *list I/O* [52] and implemented it in the PVFS [86] parallel file system. List I/O strives to reduce the number of I/O requests sent to the file server by adding native noncontiguous I/O supports the file system. Specifically, it defines a general programming interface that allows specifying an arbitrary noncontiguous I/O operation. The programming interface for a read operation is shown as follows:

pvfs_read_list (int mem_list_count, char mem_offsets[ ], char mem_lengths[ ],
int file_list_count, int file_offsets[ ], int file_lengths[ ])

Such an I/O interface is flexible in specifying noncontiguous layouts for data in both memory and files, allowing the actual I/O data and the file access information to be sent to the file server as a whole. Unlike data sieving, list I/O reduces the number of I/O requests without incurring redundant I/O data. Also, for a write operation, the read-modify-write approach in data sieving requires the I/O data to be transfered twice over the network. With list I/O, the I/O data is only transfered once for a write operation.

View I/O is another optimization technique implemented at the file system level [54]. With view I/O, a file *view* is first declare to make the noncontiguous file regions corresponding to the I/O requests visible to the application. Subsequent I/O accesses through the file view will then be mapped to those file regions accordingly. View I/O is similar to list I/O in a sense that the noncontiguous file access information is directly handled by the file system. On unique feature of view I/O is that it leverages the physical layout of file regions in servicing I/O requests, which can yield further performance improvements by reducing the indirect data exchanges. Also, unlike with list I/O, where the file access information needs to be transferred for every I/O operation, with view I/O, a file view can be reused for repeated I/O operations to save network communication costs.

Our study in Chapter 3 focuses on user-level I/O optimizations for the portability consideration. Nonetheless, our performance results in Section 3.4 shown that our asynchronous-two phase I/O approach can deliver considerable performance improvements to independent I/O enhanced with list I/O on the PVFS file system.

6.3 Remote I/O in Distributed Environments

Several efforts have been made to provide efficient remote file accesses for scientific applications through parallel I/O interfaces. RIO [87] introduced a proof-of-concept library that allows an application to access remote files with MPI-IO functions. RIO adopts a client-server architecture and it leverages the features of the ADIO [88] interface to achieve good portability across various file systems. RIO requires provisioning extra "forwarder nodes" dedicated for network communications to improve data transfer efficiency. RFS [89] is another client-server based remote I/O library. It addressed several limitations of RIO.

For instance, it removes the requirement of "forwarder nodes" and supports more updated communication protocols. RFS also reduces the visible remote I/O cost with the *active buffering* technique, which optimizes the overlap between application I/O and computation. Other approaches of translating remote I/O requests into operations of general data transferring protocols such as Grid FTP [90] and Logistic Network [91] have also been investigated. ParaMEDIC, on the other hand, focuses on aggressively reducing the amount of I/O data that needs to be shipped across the wide area network with efficient utilization of applications' semantics.

6.4 Semantic-based Data Transformation

Semantic-based data transformation is not new. Several semantic compression algorithms have been investigated in compressing relational databases [92, 93]. These algorithms first build a descriptive model based on the table semantics. With the descriptive model, the data in the database is categorized into the base data and the data that can be derived from the model. The compression can then be done by stripping out the derivable data. In the multimedia field, context-based coding techniques (similar to semantics-based approaches) have been widely used in various video compression standards [94, 95, 96]. With aid of context modeling, these techniques efficiently identify redundant information in the media. Although sharing the same goal of reducing data to store or transfer with ParaMEDIC, these data compression studies do not address the remote I/O issue.

6.5 Web Sever Scheduling

There have been numerous studies on scalable distributed web-server systems, most of which were focused on efficient request routing and assignment for content serving, as surveyed by Cardellini et al. [97]. One closely related project to our BLAST server scheduling work is the LARD scheduling algorithm [65]. LARD targets the scheduling problem on a clustered content-serving web server, with the main goal to achieve both good load balance and high locality. To take advantage of data locality, LARD maintains a map between the requested objects and the backend cluster nodes, and it strives to direct the requests to a same object to a same set of backend nodes. For skewed object access

patterns, locality-aware decision alone may result in an imbalance situation where some backend nodes are overloaded while others are underutilized. To handle this situation, LARD monitors the load (in terms of the number of active connections) on each backend node and dynamically adds or removes nodes from the node set mapped to a given object.

The LARD algorithm is not directly applicable to our BLAST server scheduling for two reasons. First, LARD assumes the backend cluster nodes are time-shared by the requests and the resource requirement is roughly the same for individual requests. With LARD, a request can be serviced immediately on a backend node chosen by the scheduling algorithm. In our target scenario, time-sharing the backend nodes is both difficult and inefficient given the closely-coupled message passing model and the intense resource requirement of servicing a BLAST request. Consequently, our PLARD algorithm in Chapter 5 adopts a space-sharing design by adding global and per-database queues to hold the unserviced queries. Second, in content-serving web services, a request can be easily handled by a single node. In a BLAST server, a request needs to be processed on multiple nodes because a BLAST search is both data- and computation- intensive. The co-scheduling requirement of compute and data resources adds extra complexity to the scheduling design.

6.6 Space-sharing Parallel Job Scheduling

Regarding space-sharing of parallel computers, a wealth of job scheduling algorithms have been proposed and evaluated, as summarized by Feitelson [98]. Space-sharing job scheduling focuses on how to partition the processors in a system and assign them to parallel jobs. There are generally four categories of partitioning approaches: *fixed partitioning*, *viable partitioning*, *adaptive partitioning* and *dynamic partitioning*.

With fixed partitioning, the system is configured into partitions according to predefined sizes. The processors in a partition are assigned to a job in an all-or-nothing manner. Fixed partitioning is easy to implement but may suffer internal fragmentation. With viable partitioning, the system allocates a partition of processors to a job according to its request. Viable partitioning avoids internal fragmentation if the architecture does not have restrictions on the partition size, but is subject to external fragmentation when the left idle processors are not enough to satisfy any job in the queue. Adaptive partitioning gives the system flexibility to choose a partition size at launching a job based on the system load

and the queue status. An example adaptive scheduling approach is *equipartition*, which strives to assign an equal number of idle processors to all the jobs in the queue. Adaptive partitioning can improve the overall system throughput but require the jobs to be *moldable*, meaning that the jobs can run on a flexible number of processors. Dynamic partitioning gives the system more flexibility by allowing the partition size allocated to a job to be changed during the runtime. However, it requires jobs to be *malleable*. A malleable job can detect the partition change and make the necessary self adjustments to continue the computation.

In practice, with the prevailing use of message passing programming interfaces such as MPI [99] and contemporary batch parallel job execution environments, adaptive or dynamic allocation of resources is rarely used on parallel computers. Instead, jobs are given the exact number of processors as requested, using strategies such as FCFS plus backfilling [100]. Our work reveals a type of real-world moldable jobs where adaptive partitioning can be applied to. In Chapter 5, we extend existing adaptive parallel job scheduling algorithms [66, 67] to the high-performance BLAST web service context.

6.7 Online Scientific Data Processing

There have been studies on hosting scientific online data processing on parallel and distributed computing resources. Wang and Mu described a distributed BLAST online service system [101], where the incoming query is assigned to the least-loaded SMP node and each node searches one entire target database. Wang et. al. introduced a service-oriented BLAST system built on peer-to-peer overlay networks [102]. This work assumes a heterogeneous environment with high communication cost. NCBI hosts a publicly accessible BLAST server on a farm of LINUX workstations [63, 64]. In the NCBI BLAST server, for a given query, the system statically splits the search into 10-20 subtasks, each searching a different piece of the database. The subtasks are scheduled independently to the machines that have just searched the same piece of data when possible. A central machine tracks and merges results from subtasks for all queries. Due to the lack of design/implementation details about the NCBI BLAST server in the literature, we were not be able to do a direct comparison. However, we argue that the NCBI server is not able to factor in the parallel efficiency by using only static task partitioning. Our scheduling approach for clustered

BLAST servers systematically optimizes scientific web services by taking into account both parallel efficiency and data locality.

Chapter 7

Conclusion

Our thesis research addressed several unique challenges in parallel and distributed genomic sequence search raised by the rapid growth of sequence data.

In Chapter 3, we considered large-scale genomic sequence search as a class of parallel applications that possesses highly irregular runtime behaviors in both computation and I/O. We found that for this type of applications, the incoordination between the I/O optimizations and the computation scheduling could result in serious performance degradations. Consequently, we proposed an integrated scheduling approach that gracefully coordinates dynamic computation load-balancing and parallel non-contiguous I/O to achieve optimized search throughput on massively parallel computers. We realized our optimizations on mpiBLAST and developed a research prototype named mpiBLAST-PIO. The experiment results on multiple platforms demonstrated that our integrated scheduling approach allows large-scale sequence search to efficiently scale on general parallel computers. In a case study on the IBM Blue Gene/P supercomputers, mpiBLAST-PIO achieved 93% parallel efficiency across 32,768 cores in solving a real-world sequence-search problem.

In Chapter 4, we presented a novel framework called "Para-MEDIC: Parallel Metadata Environment for Distributed I/O and Computing" that enables distributed sequence searching across multiple supercomputing sites connected with wide area networks. ParaMEDIC leverages application-specific semantic information to convert results data into orders-of-magnitude smaller metadata on the computing site, transfers meta data over the network and regenerates the original output on the storage site from the metadata. In other words, ParaMEDIC trades a small amount of additional computation (in the form of data

post-processing) for a potentially significant reduction in data that needs to be transferred in distributed environments. We presented the detailed design of the framework and presented experimental evaluations on different experimental as well as real distributed systems. Our results shown an *order-of-magnitude* improvement in performance for distributed parallel sequence search with ParaMEDIC in some cases.

In Chapter 5, we identified the scheduling requirements of increasingly popular parallel sequence-search web services. For our target workload, we extended and designed several adaptive scheduling strategies, namely PLARD for locality-enhancing resource partitioning, and RMAP for dynamic parallelism adjustments. These strategies automatically react to the query workload, both in terms of the request intensiveness and the data access pattern. We performed extensive performance evaluation on our scheduling algorithms on a real cluster, and our results demonstrated that PLARD can significantly reduce the amount of file I/O. Meanwhile, RMAP outperforms its static counterparts across various query workloads. Combined together, our proposed strategies can automatically configure the system for optimized query response time in various scenarios.

Bibliography

[1] G. Heffelfinger et al. Genomes to Life project proposal. http://www.genomes2life.org/SNL-ORNL-GTL-Proposal.doc.

[2] M. Marra, S. Jones, C. Astell, R. Holt, A. Brooks-Wilson, Y. Butterfield, J. Khattra, J. Asano, S. Barber, S. Chan, A. Cloutier, S. Coughlin, D. Freeman, N. Girn, O. Griffith, S. Leach, M. Mayo, H. McDonald, S. Montgomery, P. Pandoh, A. Petrescu, G. Robertson, J. Schein, A. Siddiqui, D. Smailus, J. Stott, G. Yang, F. Plummer, A. Andonov, H. Artsob, N. Bastien, K. Bernard, T. Booth, D. Bowness, M. Drebot, L. Fernando, R. Flick, M. Garbutt, M. Gray, A. Grolla, S. Jones, H. Feldmann, A. Meyers, A. Kabani, Y. Li, S. Normand, U. Stroher, G. Tipples, S. Tyler, R. Vogrig, D. Ward, B. Watson, R. Brunham, M. Krajden, M. Petric, D. Skowronski, C. Upton, and R. Roper. The genome sequence of the sars-associated coronavirus. *Science*, 2003.

[3] J. Ostell. Databases of discovery. *ACM Queue*, 3(3), 2005.

[4] D. Benson, M. Boguski, D. Lipman, J. Ostell, B. Ouellette, B. Rapp, and D. Wheeler. GenBank. *Nucleic Acids Res.*, 2002.

[5] National Center for Biotechnology Information. Genbank overview. http://www.ncbi.nlm.nih.gov/Genbank/GenbankOverview.html.

[6] Kevin Davies. Pacific biosciences preparing the 15-minute genome by 2013. http://www.bio-itworld.com/BioIT_Content.aspx?id=71746, February 2008.

[7] Ian Foster. Service-oriented science. *Science*, 308(5723), May 2005.

[8] H. Lin, X. Ma, P. Chandramohan, A. Geist, and N. Samatova. Efficient data access for parallel BLAST. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, Washington, DC, USA, 2005. IEEE Computer Society.

[9] A. Ching, W. Feng, H. Lin, X. Ma, and A. Choudhary. Exploring I/O strategies for parallel sequence database search tools with S3aSim. In *Proceedings of the International Symposium on High Performance Distributed Computing*, June 2006.

[10] O. Thorsen, K. Jian, A. Peters, B. Smith, H. Lin, and C. P. Sosa W. Feng. Parallel genomic sequence-search on a massively parallel system. In *ACM International Conference on Computing Frontiers*, 2007.

[11] H. Lin, P. Balaji, R. Poole, C. Sosa, X. Ma, and W. Feng. Massively parallel genomic sequence search on the blue gene/p architecture. In *Proceedings of the ACM/IEEE SC2008 Conference on High Performance Networking and Computing*, 2008.

[12] P. Balaji, W. Feng, and H. Lin. Semantic-based distributed i/o with the paramedic framework. In *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*, New York, NY, USA, 2008. ACM.

[13] P. Balaji, W. Feng, H. Lin, J. Archuleta, S. Matsuoka, A. Warren, J. Setubal, E. Lusk, R. Thakur, I. Foster, D. Katz, S. Jha, K. Shinpaugh, S. Coghlan, and D. Reed. Distributed i/o with paramedic: Experiences with a worldwide supercomputer. In *International Supercomputing Conference (ISC)*, 2008.

[14] H. Lin, X. Ma, J. Li, T. Yu, and N. Samatova. Adaptive request scheduling for parallel scientific web services. In Bertram Ludäscher and Nikos Mamoulis, editors, *SSDBM*, volume 5069 of *Lecture Notes in Computer Science*. Springer, 2008.

[15] H. Lin, X. Ma, J. Li, Y. T, and N. Samatova. Processor and data scheduling for online parallel sequence database servers. In *Technical Report TR-2007-23*. North Carolina State Univeristy, 2007.

[16] D. Benson, M. Boguski, D. Lipman, J. Ostell, B. Ouellette, B. Rapp, and D. Wheeler. GenBank. *Nucleic Acids Res.*, 1999.

[17] NCBI. National center for biotechnology information. http://www.ncbi.nlm.nih.gov/.

[18] DDBJ. Dna data bank of japan. http://www.ddbj.nig.ac.jp/.

[19] EMBL. European molecular biology laboratory. http://www.embl.org/.

[20] EMI. European bioinformatics institute. http://www.ebi.ac.uk/.

[21] A. Bairoch and R. Apweiler. The SWISS-PROT protein sequence data bank and its supplement TrEMBL. *Nucleic Acids Res.*, 1997.

[22] SIB. Swiss institute of bioinformatics. http://www.isb-sib.ch/.

[23] EBI. http://www.ebi.ac.uk/. http://www.ebi.ac.uk/.

[24] F. Bernstein, T. Koetzle, G.J. Williams, E.F. Meyer Jr, M.D. Brice, J.R. Rodgers, O. Kennard, T. Shimanouchi, and M Tasumi. The Protein Data Bank. A computer-based archival file for macromolecular structures. *European Journal of Biochemistry*, 1977.

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

[26] T.F. Smith and Waterman. Identification of common molecular subsequences. *J. Mol. Biol.s*, 147:195–197, 1981.

[27] Pearson WR Lipman DJ. Improved tools for biological sequence comparison. *Proc Natl Acad Sci*, 85(8):2444–2448, 1988.

[28] S. Altschula, W. Gisha, W. Millerb, E. Meyersc, and D. Lipmana. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3), 1990.

[29] S. Altschula, T. Madden, A. Schffer, J. Zhang, Z. Zhang, W. Miller, and D. Lipman. Gapped BLAST and PSI-BLAST: A new generation of protein database search programs. *Nucleic Acids Research*, 25(17), 1997.

[30] Z. Zhang, S. Schwartz, L. Wagner, and W. Miller. A greedy algorithm for aligning DNA sequences. *Journal of Computational Biology*, 7(1-2), 2000.

[31] States DJ. Gish W. Identification of protein coding regions by database similarity search. *Nat Genet.*, 3(3):266–272, 1993.

[32] Gish W. States DJ. Combined use of sequence similarity and codon bias for coding region identification. *Comput Biol.*, 1(1):39–50, 1994.

[33] W. Feng. mpiblast on the greengene distributed supercomputer. Presentation at SC05.

[34] W. Kent. BLAT - the BLAST-like alignment tool. *Genome Research*, 12(4), 2002.

[35] Mullikin JC Ning Z, Cox AJ. SSAHA: A Fast Search Method for Large DNA Databases. *Genome Res.*, 11(10):1725–1729, 2001.

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

[37] S. Schwartz, J. Kent, A. Smit, Z. Zhang, R. Baertsch, R. Hardison, D. Haussler, and W. Miller. Human-mouse alignments with blastz. *Genome Res.*, 13, 2003.

[38] M. Gardner, W. Feng, J. Archuleta, H. Lin, and X. Ma. Parallel genomic sequence-searching on an ad-hoc grid: Experiences, lessons learned, and implications. In *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing*, 2006.

[39] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[40] M. Warren and J. Salmon. A parallel hashed oct-tree n-body algorithm. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, New York, NY, USA, 1993. ACM.

[41] J. Chen and V. E. Taylor. Mesh partitioning for distributed systems: Exploring optimal number of partitions with local and remote communication. In *PPSC*, 1999.

[42] K. Schloegel, G. Karypis, and V. Kumar. Dynamic repartitioning of adaptively refined meshes. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, Washington, DC, USA, 1998. IEEE Computer Society.

[43] A. Sohn and H. Simon. S-HARP: A scalable parallel dynamic partitioner for adaptive mesh-based computations. In *Proceedings of Supercomputing 98, Orlando, Florida*, 1998.

[44] S. Hummel, E. Schonberg, and L. Flynn. Factoring: a method for scheduling parallel loops. *Commun. ACM*, 35(8), 1992.

[45] S., J. Schmidt, R. Uma, and J. Wein. Load-sharing in heterogeneous systems via weighted factoring. In *SPAA '96: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, New York, NY, USA, 1996. ACM.

[46] I. Banicescu and S. Hummel. Balancing processor loads and exploiting data locality in n-body simulations. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 43, New York, NY, USA, 1995. ACM.

[47] I. Banicescu and V. Velusamy. Load balancing highly irregular computations with the adaptive factoring. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 195, Washington, DC, USA, 2002. IEEE Computer Society.

[48] I. Banicescu, V. Velusamy, and J. Devaprasad. On the scalability of dynamic scheduling scientific applications with adaptive weighted factoring. *Cluster Computing*, 6(3), 2003.

[49] R. Thakur and A. Choudhary. An extended two-phase method for accessing sections of out-of-core arrays. *Scientific Programming*, 5(4), 1996.

[50] R. Thakur, W. Gropp, and E. Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*, May 1999.

[51] R. Thakur, W. Gropp, and E. Lusk. Optimizing noncontiguous accesses in MPI-IO. *Parallel Computing*, 28(1), January 2002.

[52] A. Ching, A. Choudhary, W. Liao, R. Ross, and W. Gropp. Noncontiguous i/o through pvfs. In *CLUSTER '02: Proceedings of the IEEE International Conference on Cluster Computing*, Washington, DC, USA, 2002. IEEE Computer Society.

[53] A. Ching, A. Choudhary, K. Coloma, W. Liao, R. Ross, and W. Gropp. Noncontiguous i/o accesses through mpi-io. In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, Washington, DC, USA, 2003. IEEE Computer Society.

[54] F. Isaila and W. Tichy. View i/o: improving the performance of non-contiguous i/o. *Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on*, pages 336–343, Dec. 2003.

[55] A. Darling, L. Carey, and W. Feng. The design, implementation, and evaluation of mpiBLAST. In *Proceedings of the ClusterWorld Conference and Expo, in conjunction with the 4th International Conference on Linux Clusters: The HPC Revolution*, 2003.

[56] H. Rangwala, E. Lantz, R. Musselman, K. Pinnow, B. Smith, , and B. Wallenfelt. Massively Parallel BLAST for the Blue Gene/L. In *High Availability and Performance Workshop*, 2005.

[57] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Standard*, July 1997.

[58] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*, February 1999.

[59] F. Schmuck and R. Haskin. GPFS: a shared-disk file system for large computing clusters. In *Proceedings of the First Conference on File and Storage Technologies*, 2002.

[60] Zfs at opensolaris.org. http://www.opensolaris.org/os/community/zfs/.

[61] D. Quintero and M. Hennecke. GPFS Multicluster with the IBM System Blue Gene Solution and eHPS Clusters. IBM Redpaper, REDP-4168-00, October 24, 2006, http://www.redbooks.ibm.com/abstracts/redp4168.html?Open.

[62] C. Sosa and G. Lakner. IBM System Blue Gene Solution: Blue Gene/P Application Development . IBM RedBook, SG24-7287, ISBN 0738488674, Rochester, Minnesoat, 2008. http://www.redbooks.ibm.com/abstracts/sg247287.html?Open.

[63] Kevin Bealer, George Coulouris, Ilya Dondoshansky, Thomas L. Madden, Yuri Merezhuk, and Yan Raytselis. A fault-tolerant parallel scheduler for blast. In *Proceedings of the Int IEEE/ACM Super Computing Conference*, 2004.

[64] S. McGinnis and T. Madden. BLAST: at the core of a powerful and diverse set of sequence analysis tools. *Nucleic Acids Res.*, 2004.

[65] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. In *ASPLOS-VIII: Proceedings of the 8th international conference on Architectural support for programming languages and operating systems*, 1998.

[66] E. Rosti, E. Smirni, L. W. Dowdy, G. Serazzi, and B. M. Carlson. Robust partitioning policies of multiprocessor systems. *Perform. Eval.*, 19(2-3):141–165, 1994.

[67] S. Dandamudi and H. Yu. Performance of adaptive space sharing processor allocation policies for distributed-memory multicomputers. *J. Parallel Distrib. Comput.*, 58(1), 1999.

[68] R. Luthy and C. Hoover. Hardware and software systems for accelerating common bioinformatics sequence analysis algorithms. *Biosilico*, 2(1), 2004.

[69] 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*, Washington, DC, USA, 1991. IEEE Computer Society.

[70] Compugen Ltd. Bioccerator. http://eta.embl-heidelberg.de:8000/, 1994.

[71] R. Braun, K. Pedretti, T. Casavant, T. Scheetz, C. Birkett, and C. Roberts. Parallelization of local BLAST service on workstation clusters. *Future Generation Computer Systems*, 17(6), 2001.

[72] N. Camp, H. Cofer, and R. Gomperts. High-throughput BLAST. http://www.sgi.com/industries/sciences/chembio/resources/papers/HTBlast/HT_Whitepaper.html.

[73] E. Chi, E. Shoop, J. Carlis, E. Retzel, and J. Riedl. Efficiency of shared-memory multiprocessors for a genetic sequence similarity search algorithm. Technical Report TR97-005, University of Minnesota, Computer Science Department, 1997.

[74] R. Bjornson, A. Sherman, S. Weston, N. Willard, and J. Wing. TurboBLAST(r): A parallel implementation of BLAST built on the TurboHub. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2002.

[75] D. Mathog. Parallel BLAST on split databases. *Bioinformatics*, 19(14), 2003.

[76] C. Oehmen and J. Nieplocha. Scalablast: A scalable implementation of blast for high-performance data-intensive bioinformatics analysis. *IEEE Trans. Parallel Distrib. Syst.*, 17(8), 2006.

[77] J. Nieplocha, R. Harrison, and R. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10(2), 1996.

[78] P. Crandall, R. Aydt, A. Chien, and D. Reed. Input/output characteristics of scalable parallel applications. In *Proceedings of Supercomputing '95*, 1995.

[79] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. S. Ellis, and Michael L. Best. File-access characteristics of parallel scientific workloads. *IEEE Trans. Parallel Distrib. Syst.*, 7(10), 1996.

[80] E. Smirni, R. Aydt, A. Chien, and D. Reed. I/O requirements of scientific applications: An evolutionary view. In *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, 1996.

[81] E. Smirni and D. A. Reed. Lessons from characterizating the input/output behavior of parallel scientific applications. *Perform. Eval.*, 33(1), 1998.

[82] R. Thakur, W. Gropp, and E. L. Lusk. An experimental evaluation of the parallel i/o systems of the ibm sp and intel paragon using a production application. In *Proceedings of the Third International ACPC Conference with Special Emphasis on Parallel Databases and Parallel I/O*, London, UK, 1996. Springer-Verlag.

[83] R. Thakur, A. Choudhary, R. Bordawekar, S. More, and S. Kuditipudi. Passion: Optimized I/O for parallel applications. *IEEE Computer*, 29(6):70–78, June 1996.

[84] D. Kotz. Disk-directed I/O for MIMD Multiprocessors. *ACM Transactions on Computer Systems*, 15(1):41–74, February 1997.

[85] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*, 1995.

[86] P. Carns, W. Ligon III, R. Ross, and R. Thakur. PVFS: A Parallel File System For Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, 2000.

[87] I. Foster, D. Kohr, R. Krishnaiyer, and J. Mogill. Remote I/O: Fast access to distant storage. In *Proceedings of the Fifth Workshop on I/O in Parallel and Distributed Systems*, pages 14–25, San Jose, CA, November 1997.

[88] R. Thakur, W. Gropp, and E. Lusk. An abstract-device interface for implementing portable parallel-i/o interfaces. In *FRONTIERS '96: Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, page 180, Washington, DC, USA, 1996. IEEE Computer Society.

[89] J. Lee, X. Ma, R. Ross, R. Thakur, and M. Winslett. RFS: Efficient and flexible remote file access for MPI-IO. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2004.

[90] T. Baer and P. Wyckoff. A parallel i/o mechanism for distributed systems. In *Proceedings of the International Conference on Cluster Computing*, 2004.

[91] J. Lee, R. Ross, S. Atchley, M. Beck, and R. Thakur. MPI-IO/L: efficient remote i/o for mpi-io via logistical networking. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006*, 2006.

[92] B. Shivnath, M. Garofalakis, and R. Rastogi. Spartan: A model-based semantic compression system for massive data tables. In *International Conference on Management of Data (SIGMOD 2001)*, May 2001.

[93] H. V. Jagadish, J. Madar, and Raymond T. Ng. Semantic compression and pattern extraction with fascicles. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[94] D. Marpe, H. Schwarz, and T. Wiegand. Context-based adaptive binary arithmetic coding in the h.264/avc video compression standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 13(7), July 2003.

[95] Coding of Audio-Visual Objects - Part 2: Visual. ISO/IEC JTC1, ISO/IEC 14 496-2 (MPEG-4 Visual version 1), Apr. 1999; Amendment1 (version 2), Feb. 2000; Amendment 4 (streaming profile), Jan. 2001.

[96] T. Wiegand D. Marpe, H. Schwarz. Context-based adaptive binary arithmetic coding in the h.264/avc video compression standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 13(7), 2003.

[97] V. Cardellini, E. Casalicchio, M. Colajanni, and P. Yu. The state of the art in locally distributed web-server systems. *ACM Computing Surveys*, 34(2), 2002.

[98] D. Feitelson. A survey of scheduling in multiprogrammed parallel systems. Technical Report IBM/RC 19790(87657), 1994.

[99] Message Passing Interface Forum. *MPI: Message-Passing Interface Standard*, June 1995.

[100] A. Mu'alem and D. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12, 2001.

[101] J. Wang and Q. Mu. Soap-HT-BLAST: high throughput BLAST based on Web services. *BIOINFORMATICS -OXFORD-*, 2003.

[102] C. Wang, B. Alqaralleh, B. Zhou, M. Till, and A. Zomaya. A BLAST service built on data indexed overlay network. *e-science*, 2005.