A Pluggable Framework for Parallel Pairwise Sequence Search

Jeremy Archuleta, Wu-chun Feng, Eli Tilevich

Abstract— **The current and near future of the computing industry is one of multi-core and multi-processor technology. Most existing sequence-search tools have been designed with a focus on single-core, single-processor systems. This discrepancy between software design and hardware architecture substantially hinders sequence-search performance by not allowing full utilization of the hardware.**

This paper presents a novel framework that will aid the conversion of serial sequence-search tools into a parallel version that can take full advantage of the available hardware. The framework, which is based on a software architecture called mixin layers with refined roles, enables modules to be plugged into the framework with minimal effort. The inherent modular design improves maintenance and extensibility, thus opening up a plethora of opportunities for advanced algorithmic features to be developed and incorporated while routine maintenance of the codebase persists.

I. Introduction

Every twelve to eighteen months the collective amount of genetic information doubles [1], [2]. While beneficial to our knowledge of genetics, this increased volume of information doubles the amount of computation required when comparing an unknown sequence to the databases of known sequences. For the last couple of decades, the computer industry has been able to thwart this data explosion by doubling the performance of the processor, i.e., central processing unit (CPU), every eighteen months through improvements in serial processing. However, after hitting the proverbial wall for improving serial CPU performance, additional performance gains must be achieved through parallel processing, i.e., multi-core and multi-CPU technology.

This shift in CPU technology will affect nearly all existing sequence-search tools. These tools, which have been invaluable to bioinformaticists, are designed to run only on a single CPU core and are thus unable to take advantage of multiple cores and multiple CPUs. In order to maintain the performance needed to keep pace with the growth of genetic sequence databases, sequence-search tools must be parallelized in order to take advantage of emergent architectures that use multiple cores and multiple CPUs.

There are two main approaches to transform sequence searching from a sequential environment to a parallel environment: (1) run multiple sequential searches in parallel or (2) modify the serial search algorithm to become a parallel algorithm.

Although executing multiple sequential searches in parallel increases throughput by an amount that is proportional to the number of simultaneous sequential searches, the latency of each search will continue to increase at the rate that sequence databases are growing (i.e., exponentially). So, although there may be hundreds of simultaneous searches running, each search will take longer and longer to complete. Furthermore, unless the number of simultaneous searches increases proportional to the growth of the sequence database size, throughput will actually decrease.

Transforming a serial algorithm into a parallel one, however, requires knowledge in parallel programming, an area that has traditionally been confined to high-performance computing. Because sequence-search tools were not originally designed with parallelism in mind, it would be unfair to impose on the maintainers of these search tools the burden of converting these tools into efficient parallel sequence-search tools. Moreover, the effort in converting each search tool from serial to parallel would involve reinventing the wheel over and over again because many of the search tools follow a "scatter-search-gather" execution flow when parallelized.

In this paper, we present a novel framework that will vastly improve the process of parallelizing sequential-search tools and improve the performance of existing search tools. By utilizing this framework, pairwise sequence-search tools can be "plugged in" with minimal effort and be immediately transformed into efficient parallel search tools.

The rest of this paper is structured as follows. Section II provides background on existing parallelization strategies for pairwise alignment. Section III describes our novel parallelization framework. Finally, we conclude with some preliminary results and future work in Section IV.

II. Existing Strategies

There has been no shortage of effort in parallelizing sequence-search tools [3], [4], [5], [6], [7]. The basic parallelization approach that most schemes utilize is "master-worker" with an execution flow of scatter, search, and gather. In this manner, the master process scatters relevant search input (e.g., database sequences, unknown query sequences, parameters, etc.) to each worker process. Each worker process performs a search using the received search input, and the master process gathers the results. However, even with such a basic strategy, there exist several variations.

A. Segmentation and Fragmentation

The key part of the scatter phase is knowing *what* to scatter — the query or the database. Query segmentation (QS) [4], [5], [6], [7] partitions an aggregate query into individual subqueries and scatters each subquery (along with the entire database to be searched) to each worker, as shown in Fig. 1.

J. Archuleta, W. Feng and E. Tilevich are with the Department of Computer Science, Virginia Tech, Blacksburg, VA 24061, USA (jsarch, feng, tilevich)@cs.vt.edu

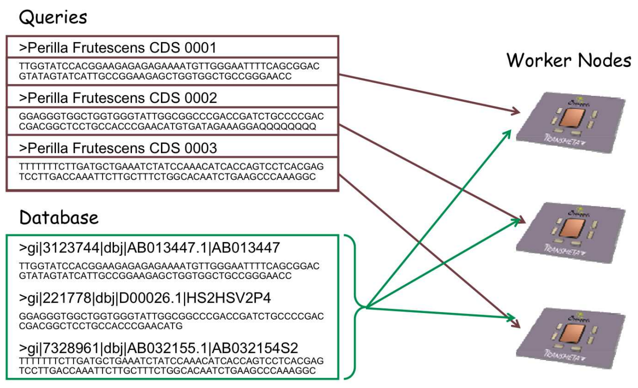
Queries
>Perilla Frutescens CDS 0001
>Perilla Frutescens CDS 0002
>Perilla Frutescens CDS 0003
Database
>gi|3123744|dbj|AB013447.1|AB013447
>gi|221778|dbj|D00026.1|HS2HSV2P4
>gi|7328961|dbj|AB032155.1|AB032154S2
Worker Nodes

Fig. 1. Overview of Query Segmentation (QS)

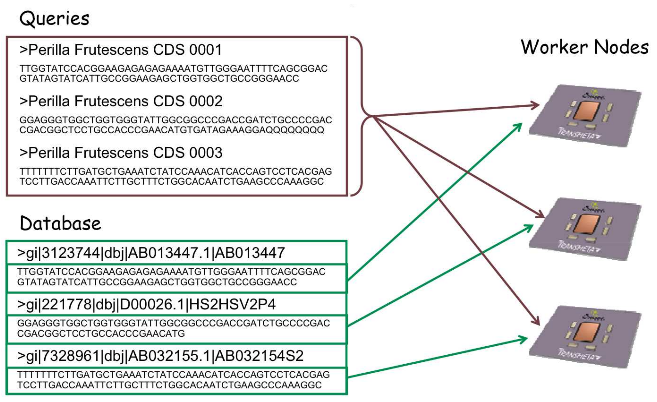
Queries
>Perilla Frutescens CDS 0001
>Perilla Frutescens CDS 0002
>Perilla Frutescens CDS 0003
Database
>gi|3123744|dbj|AB013447.1|AB013447
>gi|221778|dbj|D00026.1|HS2HSV2P4
>gi|7328961|dbj|AB032155.1|AB032154S2
Worker Nodes

Fig. 2. Overview of Database Fragmentation (DF)

However, QS is not without a substantial issue: each subquery requires a complete copy of the entire database and the database is oftentimes much larger than the size of memory. This puts tremendous pressure on the system hardware to provide each worker process with a copy of (or at the very least, access to) the entire database. The pressure can be alleviated by requiring that each worker process have enough local storage to store the entire database or by investing in a globally accessible parallel filesystem. However, paging the database from storage kills performance as it is orders of magnitude slower than accessing the database from memory. Thus, QS alone is a parallelization strategy that will not scale as databases continue to double in size every 12 to 18 months [1], [2].

Rather than scattering the query, database fragmentation (DF) scatters the database, as first implemented in mpiBLAST [3] — a parallelization of the National Center for Biotechnology Information Basic Local Alignment Search Tool, commonly known as NCBI BLAST [8], [9]. Specifically, DF splits the database into fragments that fit into each worker's memory but then delivers the same query to each worker, as shown in Fig. 2.

Using DF results in super-linear speedup because each database fragment now fits in memory, thus eliminating the need to page data between memory and disk. For over a decade, this novel technique of DF was presumed to be impossible for NCBI BLAST because generating correct "e-value" scores depended on each sequence knowing the "effective size" of the entire database. However, since the e-value is calculated mathematically, mpiBLAST surmounted this "impossible" obstacle by appending the relevant information to the database fragment, thus allowing each worker process to calculate exact e-value scores.

Since query segmentation (QS) and database fragmentation (DF) are orthogonal parallelization strategies, we combined the approaches recently, i.e., QS+DF. This hybrid approach to parallelizing BLAST realized the benefit of both fitting each fragment in memory through DF and executing subqueries in parallel via QS. In this way, throughput is increased through QS by simultaneously executing many subqueries, and latency is decreased via DF by eliminating paging from storage. The result is a 305-fold speedup with mpiBLAST version 1.4.0, as seen in the following table.

# Workers	DF	QS+DF
8	23	23
16	45	48
32	81	76
64	147	173
128	187	305

TABLE I
SPEEDUP OF DF VS. DF+QS (BASELINE = 1 WORKER)

B. Parallelization Model

The parallelization model often used in parallel sequence searching is known as a "master-worker" model. The master process is responsible for distributing jobs to worker processes, usually one worker process per CPU (or CPU core). This separation of tasks appears to be well suited to the embarassingly parallel nature of DF and QS. For example, a query is segmented into Q subqueries, and each subquery is assigned to a group of N worker processes. Within each group of worker processes, the database is split into M fragments, where usually $M = N$. Thus, each worker process searches some portion of the query against some portion of the database. After finishing a search, the worker process notifies the master process of completion, and the master gathers all the results to process for output. Unfortunately, as the number of workers increases, a bottleneck at the master (during the gather phase) begins to dominate the execution time [10].

Alleviating this bottleneck has been the subject of several papers [10], [11], [12]. From these papers, what has become clear is that the single-master, multiple-worker parallelization model will not scale. As seen in [11] and [12], a multiple-master approach improves parallel BLAST scalability to thousands of processors. With this in mind, a key feature of our new framework is the ability for various parallelization models (including the multiple-master strategy) to be implemented without necessitating wholesale changes to the framework.

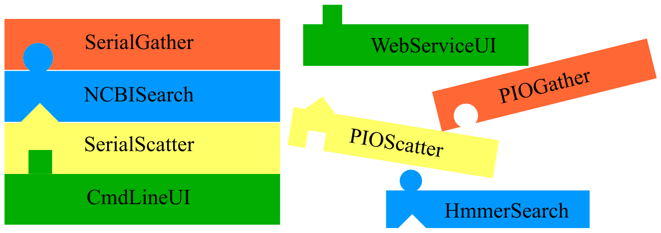

Fig. 3. Example of Interchangeable Modules

III. The Framework

In presenting our framework, we focus on the main design and implementation decisions that we made and their relevance to bioinformatics. We elide the low-level implementation details as they are not pertinent to the discussion and are beyond the scope of this paper.

A. Features

Our framework uses state-of-the-art software engineering tools and techniques to provide a plug-in replaceable infrastructure for sequence-search tools. It enables sequence-search tools to take advantage of the parallel hardware both in everyday personal computers as well as in specialized high-performance computing systems. In addition, our framework is easy to extend and maintain.

1) Modularity: The crux of the framework is modularity, which enables maintainable and extensible high-performance sequence search. Fig. 3 shows a subset of the modules of the framework, with each phase of a search algorithm confined to a separate module. The modularity based on the phases of the algorithm makes it possible to interchange modules independently of each other. This ability to mix-and-match modules is essential in creating search tools that can take full advantage of different hardware and software configurations in modern parallel systems (e.g., parallel NCBI BLAST on a multi-core PC, parallel HMMER on IBM BlueGene with parallel input/output performance enhancements, etc.)

2) Extensibility: One of the key design objectives that we had for our framework was that it be easy to extend. As pointed out earlier, high modularity is conducive to flexible interchanges of different phases of a search algorithm in a plug-and-play fashion, allowing for an extensive feature list. New search algorithms, new parallelization models, and novel performance-enhancement modules are all examples of the features that can be quickly incorporated into this framework, as exemplified in Fig. 3.

Using our framework hides away many of the low-level nuances of parallel programming (e.g., creation and scheduling of tasks), which traditionally can be a hurdle in parallelizing a serial algorithm. This abstraction allows developers to focus on the details of porting the search algorithm to the framework, which primarily involves modifying the algorithm to obtain input data from the "database" module and to return output data to the "writer" module.

The scheduler is the brain of our framework and handles what, when, and to whom the processes communicate. In the framework, the "scheduler" module provides all the scheduling functionality. When the default master-worker scheduler is not appropriate for a particular system, such as in a massively-parallel system like IBM BlueGene/L, an alternative state-of-the-art parallelization model such as [11], [12] can be easily provided via a new scheduler.

In addition to replacing the scheduler, specialized input-output modules can further improve performance. For example, one can provide a special-purpose "write" module utilizing a parallel filesystem with parallel input-output features available, a la [10].

3) Maintainability: The modular design of our framework improves maintainability. With functionality encapsulated within modules, bugs are likely to be constrained to individual modules. For example, an error in the command-line processing would be found in the command-line module and not in the formatting module. Thus, routine maintenance of the framework is orthogonal to its development; a bug can be tracked down, fixed, tested, and the fix deployed independently of feature development.

In addition to modularity, strict adherence to the ANSI C++ syntax and unit testing further improve maintainability. The primary advantage of using ANSI C++ syntax is maximal portability as nearly all major platforms support ANSI C++. Unit testing allows each "unit" of code (i.e., module) to be tested thoroughly and rigorously in the absence of the entire application [13]. Our framework is a good fit for unit testing due to the low coupling it exhibits. Additional details of the software engineering rationale behind our framework can be found in [14].

B. Implementation

The software architecture that enables all the above-mentioned benefits for this framework uses mixin layers as its implementation mechanism. Mixin layers is an implementation technique for collaboration-based designs, which assembles software modules in layers in which each successive layer is represented as a collection of inner classes [15], [16], [17]. Furthermore, both the enclosing class and its inner classes participate in an inheritance relationship with an abstract superclass: the enclosing class inherits from the enclosing superclass, and each inner class inherits from its corresponding inner class in the super enclosing class.

This design enables flexibility in adding functionality with each layer: a layer defines inner classes only for those objects for which it needs to add functionality. A typical C++ implementation of a mixin layer is as follows:

```
template <class Super>
class MixinLayer : public Super {
 class Inner1 : public Super::Inner1 { /* body */ };
 class Inner2 : public Super::Inner2 { /* body */ };
};
```

Each inner class represents a different "role" in the design. We call our implementation "mixin layers with refined roles," as it enables flexible collaboration between different roles without duplicating code. Specifically, the roles in our framework are *Common*, *Master*, and *Worker*,

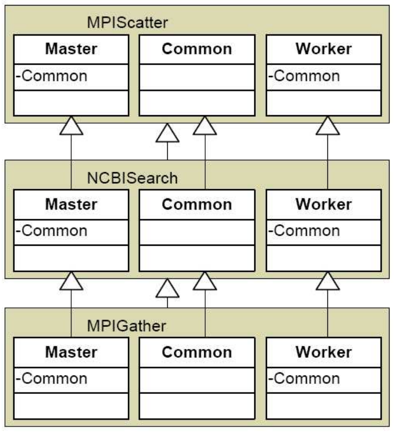

Fig. 4. Mixin Layers with Refined Roles

with *Common* representing common functionality used by both *Master* and *Worker*. Fig. 4 demonstrates the main details of our implementation, showing how it provides the benefits of maintainability and extensibility. With respect to maintenance, code reuse greatly reduces the possibility of *Master* and *Worker* from having different (and incompatible) implementations of the same functionality. In terms of extensibility, the separate classes allow both *Master* and *Worker* to be enhanced independently.

We arrived at the decision to use "mixin layers with refined roles" as the architectural design for our framework after a thorough analysis of multiple architectural styles. The details of this analysis are beyond the scope of this paper; the interested reader can learn about our experiences in [14]. However, the primary reason for using "mixin layers with refined roles" is that this design provides maintainable and extensible high-performance through plug-in replaceable and reusable software components that can be easily mixed and matched. One can think of the different layers as Lego^{TM} blocks for constructing a parallel search application, with the shape of a block determining with which other blocks it can be combined to ensure an efficient parallel execution.

IV. Future Work and Conclusions

This framework has been utilized to create mpiBLAST-2.0, a parallelization of the new NCBI BLAST C++ Toolkit. Although mpiBLAST-2.0 improves overall execution time of searching *E. chrysanthemi* against the *NT* database in relation to mpiBLAST-1.4.0, e.g., by 43% on 32 nodes of SystemX [18], it is difficult to quantify precisely what percentage of this improvement is due to the new design and what is due to the search algorithm. We plan to thoroughly evaluate the framework in an effort to minimize the overhead of the framework such that the framework will scale to hundreds and thousands of processor cores.

We also plan to implement many of the enhancements mentioned in this paper such as multiple master, parallel input-output, and other search algorithms. These enhancements have already been proven to be useful in [11] and [12] and would be well worth incorporating into the framework.

Additionally, improvements to sequence search should not be confined to algorithms and performance. The user interface (UI) of sequence search tools is an area of sequence searching that has been relatively ignored due to search execution times being on the order of minutes, if not hours. Harnessing the parallel processing capability of current and future generations of computers, sequence search is set to become an interactive experience. Through this framework, research and development focusing on the UI is now possible and creating a new UI is as easy as replacing a layer.

In conclusion, parallel processing has become a necessity for sequence search tools, and we believe that our framework will facilitate the creation of parallel tools in this important domain.

References

[1] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, B. A. Rapp, and D. L. Wheeler, "GenBank," *Nucleic Acids Res.*, vol. 30, pp. 17–20, 2002.
[2] "Gold - Genomes Online Database," http://www.genomesonline.org/.
[3] A. Darling, L. Carey, and W. Feng, "The Design, Implementation, and Evaluation of mpiBLAST," in *International Conference on Linux Clusters: The HPC Revolution 2003*, 2003.
[4] R. Bjornson, A. Sherman, S. Weston, N. Willard, and J. Wing, "TurboBLAST(r): A parallel implementation of BLAST built on the TurboHub," in *International Parallel and Distributed Processing Symposium*, Apr 2002.
[5] N. Camp, H. Cofer, and R. Gomperts, "High-throughput BLAST," SGI, Tech. Rep., Sep 1998.
[6] K. Pedretti, T. Casavant, R. Braun, T. Scheetz, C. Birkett, and C. Roberts, "Three Complementary Approaches to Parallelization of Local BLAST Service on Workstation Clusters," *Lecture Notes in Computer Science*, vol. 1662, pp. 271–282, 1999.
[7] E. Chi, E. Shoop, J. Carlis, E. Retzel, and J. Riedl, "Efficiency of Shared-Memory Multiprocessors for a Genetic Sequence Similarity Search Algorithm ," University of Minnesota, Tech. Rep., 1997.
[8] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, "Basic Local Alignment Search Tool," *Journal of Molecular Biology*, vol. 215, pp. 403–410, 1990.
[9] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman, "Gapped BLAST and PSIBLAST: A New Generation of Protein Database Search Programs," *Nucleic Acids Research*, vol. 25, pp. 3389–3402, 1997.
[10] H. Lin, X. Ma, P. Chandramohan, A. Geist, and N. Samatova, "Efficient Data Access for Parallel BLAST," in *International Parallel and Distributed Processing Symposium*, Apr 2005.
[11] C. Oehmen and J. Nieplocha, "ScalaBLAST: A Scalable Implementation of BLAST for High-Performance Data-Intensive Bioinformatics Analysis," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 8, pp. 740–749, 2006.
[12] O. Thorsen, K. Jiang, A. Peters, B. Smith, H. Lin, W. Feng, and C. Sosa, "Parallel Genomic Sequence-Search on a Massively Parallel System," in *ACM International Conference on Computing Frontiers*, May 2007.
[13] K. Beck, *Test-Driven Development: By Example*. Addison-Wesley, 2003.
[14] J. Archuleta, E. Tilevich, and W. Feng, "A Maintainable Software Architecture for Fast and Modular Bioinformatics Sequence Search," in *IEEE International Conference on Software Maintenance*, Oct 2007.
[15] Y. Smaragdakis and D. Batory, "Implementing Layered Designs with Mixin Layers," in *European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag LNCS 1445, 1998.
[16] ——, "Mixin-Based Programming in C++," in *Generative and Component-Based Software Engineering Symposium (GCSE)*, 2000.
[17] ——, "Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs," *ACM Transactions on Software Engineering and Methodologies (TOSEM)*, vol. 11, no. 2, pp. 215–255, 2002.
[18] "Advanced Research Computing," http://www.arc.vt.edu.