HSPp-BLAST: Highly Scalable Parallel PSI-BLAST for Very Large-scale Sequence Searches

Bhanu Rekepalli, Aaron Vose and Paul Giblock
Joint Institute for Computational Sciences
University of Tennessee and Oak Ridge National Laboratory
Oak Ridge, TN, 37831, USA
brekapal@utk.edu, avose@eecs.utk.edu and pgiblock@utk.edu

Abstract – Based on recent published articles, the growth of genomic data has overtaken and outpaced both performance improvements of storage technologies and processing power due to the revolutionary advancements of next generation sequencing technologies. By bringing down the costs and increasing throughput by many orders of magnitude with sequencing technologies, data is doubling every 9 months resulting in the exponential growth of genomic data in recent years. However, data analysis becomes increasingly difficult and can be prohibitive, as existing bioinformatics tools developed in the past decade focus mainly on desktops, workstations and small clusters that have limited capabilities. Improving the performance and scalability of such tools is critical to transforming ever-growing raw genomic data into biological knowledge containing invaluable information directly related to human health. This paper describes a new software application which includes optimization techniques improving the scalability of a most widely used bioinformatics tool "PSI-BLAST" on advanced parallel architectures, pushing the envelope of biological data analysis. We show that our improvements allow near-linear scaling to tens of thousands of processing cores, up to the maximum non-capability size on current petaflop supercomputers. This new tool increases by 5 orders of magnitude the amount of genomics data that can be processed per hour.

1 Introduction

The novel genomic data generated by sequencing machines are processed through various bioinformatics tools to become annotated and deposited into databases. Most of these tool packages consist of a series of sequence similarity search tools for annotation, since sequence similarities may be the consequence of structural, functional, and evolutionary relationships between the sequences. From the alignment of two sequences one can infer the evolutionary relationship, functional domains shared between proteins, and transcription-factor binding sites for DNA sequences. A most widely used tool for such comparisons is Basic Local Alignment Search Tool (BLAST) [1,2]. There are many implementation of the BLAST algorithm with the implementation by the National Center for Biotechnology Information (NCBI) being the most popular. Also there are many programs within NCBI BLAST for both nucleotide and protein sequence similarity searches.

Nucleotide sequences have only four bases (ATGC), whereas protein sequences consist of 20 amino acids (AAs), thus resulting in a larger variety of sequence characters with increased complexity which makes it easier to detect patterns of sequence similarity between protein sequences when compared to DNA sequences [3]. Thus protein sequence database searches yield more significant matches when compared to DNA sequence databases for a specific protein sequence [4]. This paper focuses on one particular protein sequence search tool known as Position Specific Iterative BLAST (PSI-BLAST) [2] as it is a most widely used tool known for its sensitivity and robustness. PSI-BLAST uses the gapped protein search program known as BLASTP for searching the query protein sequence against the protein database. The first iteration of PSI-BLAST is with the standard substitution matrix, a matrix containing values proportional to the probability that one amino acid is replaced by another amino acid for all pairs of amino acids [1, 2]. Once proteins similar to the query sequence (known as relatives) are found, PSI-BLAST constructs a profile and multiple alignments based on these relatives. This profile is then compared to the protein database to seek local alignments using the BLASTP program. In the second iteration, once the local alignments are constructed, PSI-BLAST estimates their statistical significance to find new relatives. Now a new profile is generated and PSI-BLAST iterates using this new profile. The process is repeated for a given number of iterations or until no new relatives or protein sequence matches are found thus reaching convergence [2,5].

These PSI-BLAST runs are both computationally intensive and data intensive operations taking anywhere from a few seconds to tens of minutes based on the size of the query sequences and the size of the database against which the query is searched. There are many implementations of the parallelized BLAST tool using either BLASTp or BLASTn functions in particular. These parallelizations incorporate either database partitioning techniques such as mpiBLAST [6] and pioBLAST [7] or query sequence partitioning techniques as seen in other implementations [8,9] or use MapReduce-MPI library to split work [10]. Previous work on IBM's Blue Gene/L even demonstrated

scalability up to 32,768 processors combining both techniques simultaneously [11]. But due to the iterative nature and complexities in the parallelization of PSI-BLAST, there are no effective parallel implementations for PSI-BLAST. Thus there is an urgent need to parallelize PSI-BLAST to keep up with the exponential growth [12] of genomic data. In our approach we used a combination of threads and MPI to parallelize PSI-BLAST to tens of thousands of cores, simultaneously retaining the core original functionality of the BLAST code on the Kraken supercomputer. We also identify a number of important performance issues, and demonstrate that our improvements allow near-linear scaling to 48,304 cores and beyond. Specifically, we find the following components to be critical for effective scalability of PSI-BLAST: 1) efficient database distribution; 2) intelligent, hierarchical, dynamic load balancing; and 3) high-throughput buffered parallel I/O of resultant data, as described in detail in the following sections.

2 Methods

2.1 Wrapping of NCBI BLAST

During the process of improving the scalability of PSI-BLAST, great care was taken to ensure that our updated version of NCBI's tool can be trusted to produce accurate and correct results. Additionally, the NCBI code base is quite large, on the order of 1.3 million lines of source code; it is desirable to make as few changes as possible to this massive collection of code [13]. To this end, we create a wrapper for NCBI's PSI-BLAST tool *blastpgp* which requires a very small number of changes to the original code. This wrapper serves to handle the inputs and outputs of PSI-BLAST in an efficient manner, leaving the core of the application untouched. This is accomplished through the use of a custom I/O library named *stdiowrap* to redirect the inputs and outputs directly to and from the *blastpgp* program, as well as a hybrid MPI / threaded wrapper named *mcw* (for the three levels of its hierarchical design: master, controller, and worker) to manage job control and I/O on the large scale. Together, the small scale wrapper *stdiowrap* and large scale wrapper *mcw* allow the existing NCBI PSI-BLAST to scale to core counts never before achievable.

2.2 Wrapper: *stdiowrap*

The *stdiowrap* library which directly interacts with NCBI's code implements a subset the C language's standard input and output routines, as found in the common *stdio.h* include file. When our replacement I/O file *stdiowrap.h* is included in the NCBI source code and the NCBI source is linked against the *stdiowrap.o* object, the inputs and outputs of *blastpgp* are automatically redirected to operate on a number of shared memory segments in RAM, rather than with files which would traditionally be located on disk. This allows the NCBI code to continue to operate as if it was working with files, while all traditional disk operations now take place in memory instead. This is a critical component of our optimization, as operations on memory are significantly faster than those on disk. While *stdiowrap* manages the interaction between PSI-BLAST and the shared memory segments holding input and output data, it is up to the MPI / threaded wrapper *mcw* to then manage interaction between the shared memory segments and the actual filesystem.

2.3 Wrapper: *mcw*

The large-scale wrapper is named *mcw* for the three levels of its hierarchical design: Master, Controller, and Worker. Its primary job is to efficiently distribute the sequence database and query sequences to those in need, to setup and manage the shared memory segments that are used by PSI-BLAST through the *stdiowrap* interface, and to quickly process the resultant output data from the search processes. The tasks of *mcw* are accomplished through the use of a hierarchical design with three levels as depicted in Figure 1 and described in detail in later sections.

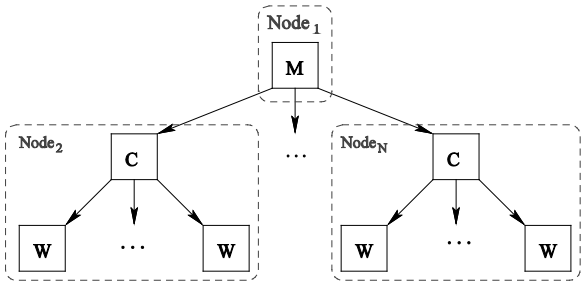

Figure 1: The **M**aster distributes blocks of query sequences to **C**ontrollers which distribute to **W**orkers which run PSI-BLAST.

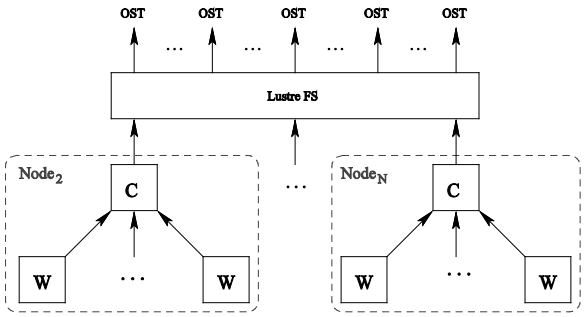

Figure 2: The **W**orkers send result data to **C**ontrollers which compress and buffer the data before writing it to disk in parallel.

2.4 Master

The master is positioned at the top of the hierarchy and is responsible for distributing the sequence database as well as the query sequences. Use of a single master is desired, as long as the master is able to keep up with the demand for work from the subordinates in need. This is because it is typically wasteful to use processing time on managerial processes; CPU time is better spent on the core PSI-BLAST computation. When a job first starts, the needed sequence database is loaded from disk by the

```
while (1):
  request    = get_work_request_from_controllers();
  num_blocks = request.num_requested *
               PREFETCH_AGGRESSIVENESS;
  work_unit  = get_next_sequence_blocks(num_blocks );
  if( work_unit == NO_MORE_WORK ) {
    break;
  }
  send_work_to_controller(request.controller,
                          work_unit);
```

Figure 3: Pseudocode showing high-level behavior of the Master process operating as an on-demand work server.

master into memory and then broadcast to all nodes in need through a single collective MPI call. This distribution is very efficient and scales logarithmically in the number of nodes. After the database has been sent to the nodes, the master begins to act as a work distribution server, where work is sent as precompressed blocks of FASTA query sequences [14]. The controllers, occupying the level just below the master in the hierarchy, will request work units from the master when they desire more work. The master then responds to requests for work and forwards blocks of compressed work units to the requesting controllers. This on-demand request / response architecture helps to ensure an even load balance as work units are only sent to controllers which need them. Additionally, to help reduce load imbalance toward the end of the job when some cores may be idle while others are finishing the last bits of work, the master transitions from sending longer query sequences at the start of a job to sending shorter query sequences toward the end of a job. This sequence distribution order is utilized because longer sequences take more time to process, as well as differ more in processing time than shorter sequences in the first iteration of PSI-BLAST. The high-level operation of the master during the distribution of work units is depicted in Figure 3.

2.5 Controller

The controllers occupy the middle level of the hierarchy and mediate between the workers on one hand and the master and filesystem on the other. Each controller is composed of two threads, 1) the controller thread which mediates between the workers and master and 2) a writer thread which is responsible for processing the resultant data from the workers and ultimately writing the data to disk. The controller thread asks for work from the master on behalf of the worker threads, and tries to keep its local work queue full for the workers. After the initial startup period when controllers fill their local work queue, the controllers try to keep their local work queue full by prefetching from the master when needed so that the worker threads never have to wait for additional work.

After the initial startup period, the goal is hide all latency of interaction with the master from the worker threads and PSI-BLAST computations. This managerial role of

```
while (1):
  request = get_work_request_from_workers();
  if( workbuffer.num_queued < NCORES ) {
    num_blocks = NCORES - workbuffer.num_queued;
    work_unit =
            get_work_blocks_from_master(num_blocks);
    if( work_unit == NO_MORE_WORK ) { break; }
    add_work_to_buffer(workbuffer, work_unit);
  }
  send_work_to_worker(
          take_work_from_buffer(workbuffer));
```

Figure 4a: Pseudocode showing (a) the work management process employed by the Controller threads

```
while(1):
  output = wait_for_output_from_any_BLAST();
  add_output_to_uncompressed_buffer(output);
  if( is_full(uncompressed_buffer) ) {
    coutput = compress(uncompressed_buffer);
    add_output_to_compressed_buffer(coutput);
    clear_buffer(uncompressed_buffer);
    if( is_full(compressed_buffer) ) {
      write(compressed_buffer);
      clear_buffer(compressed_buffer);
    }
  }
```

Figure 4b: Pseudocode showing the two-stage buffering process employed by the writer thread accompanying each Controller.

the controller is depicted in Figure 4a.

When the worker threads produce resultant data from the PSI-BLAST search, they pass this data to the writer thread of the controller on the local node, by performing a simple *memcpy()* operation. This writer thread employs a two-stage buffering process whereby the workers fill the first buffer with the resultant uncompressed XML data as output by NCBI's PSI-BLAST process. We chose the XML output format as it contains all the required data for complete analysis when compared to other output formats that do not include all available data from the PSI-BLAST search process. When the first buffer holding uncompressed data is full, it is compressed and moved to the second buffer which holds compressed data in the form of "records". Each record consists of a four byte size field followed by a compressed block of XML data with a length matching that in the preceding size field. Once this second buffer holding compressed data is full, it is flushed to disk in parallel utilizing a distributed filesystem if present. This process is depicted in Figure 2 with pseudocode in Figure 4b. Additionally, the output files to which data is written are arranged in a hierarchical directory structure which allows parallel use of the object storage targets which form the back end of the distributed Lustre filesystem used on Kraken [15,16]. Finally, the output files were configured with the optimal stripe count of one, as a file-per-processes output scheme is utilized. While some of

these enhancements were designed with the Luster filesystem in mind, most should be compatible with other distributed and networked filesystems as well.

2.6 Worker

The workers are threads occupying the lowest level of the hierarchy. These threads are responsible for acquiring work from their local controller, installing this work in the form of input query sequences into the appropriate shared memory segments on their node, and launching the actual PSI-BLAST application, *blastpgp*. When the PSI-BLAST process is finished processing its input query sequences and writing the result to the output shared memory segment, the worker then copies the resultant data from the shared memory segment into the output buffer of the writer thread on the node. This architecture allows the PSI-BLAST computation to proceed continuously as long as the controller thread is able to keep the local work queue full by prefetching from the master and the writer thread is able to flush the resultant data to disk by means of parallel I/O to a distributed filesystem such as LustreFS. The high-level operation of the workers is depicted in the Figure 5.

```
while (1):
  work_unit = get_work_block_from_controller();
  if( work_unit == NO_MORE_WORK ) {
    break;
  }
  output = run_blast_search(work_unit);
  copy_data_to_writer_thread(output);
```

Figure 5: Pseudocode showing the high-level operation of the Worker threads.

2.7 Memory Subsystem Enhancements

In addition to the *stdiowrap* and *mcw* wrappers, a number of other enhancements were investigated which provide the PSI-BLAST process with additional speed improvements. First, due to the limited amount of RAM available on each compute node of the Kraken supercomputer (16 GB), making the most efficient use of this memory is critical. To this end, we store a single copy of the sequence database per node in a shared memory segment for the duration of all sequence searches performed on the node, unlike the serial version of PSI-BLAST which can load the database from disk multiple times. Further, this single shared copy of the database allows all PSI-BLAST processes on a given node access to the same database, providing a twelve times increase in effective memory capacity for holding the sequence database.

Additionally, extensive profiling and code tracing efforts performed on NCBI's *blastpgp* software show that the PSI-BLAST processes memory maps the sequence database into its virtual address space and then scans the sequence database in memory in a roughly linear fashion from low addresses to high addresses. This behavior results in the database being loaded into memory from disk through a large number of page-faults as the operating system loads pages one at a time into memory on behalf of the user processes as they accesses different areas of the sequence database. This page-fault based loading mechanism is suboptimal as it typically results in many tiny reads to the filesystem. This behavior is depicted in Figure 6a.

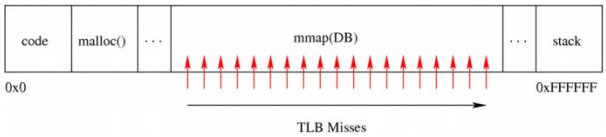

Figure 6a: The default page size results in many TLB misses as the database is scanned in a linear fashion. Worse, the default implementation can cause not only a TLB miss, but a page fault on every 4 KB page boundary as well.

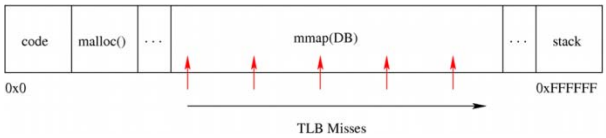

Figure 6b: 2 MB pages drastically reduce the number of TLB misses, and preloading the database into memory also solves the issue of slow page-fault based loading. In this figure, the two output buffers used by the controller are part of the *malloc()* section.

Further, by preloading the database into a shared memory segment all at once through the use of a single system call, this page-fault based loading scheme is avoided. Also, the shared memory segments used to hold the sequence database can be backed by 2 MB large pages rather than the default 4 KB pages. This enhancement better utilizes the Translation Look-aside Buffer (TLB), a hardware cache which translates virtual program addresses into absolute physical hardware addresses. When using the default small page size, the linear scan can cause a TLB cache miss on every 4 KB boundary. By utilizing larger 2 MB pages, TLB cache misses can be drastically reduced, as depicted in Figure 6b. In our testing, the use of large pages can reduce the processing time of PSI-BLAST by 15% to 60% in a number of real world cases, depending on the search parameters used.

2.8 Output I/O Enhancements

Past work with the *mcw* wrapper and a single iteration of PSI-BLAST has highlighted a number of I/O schemes which can significantly affect the performance of output I/O. Our original *mcw* version used an on-demand output scheme, where resultant XML data was processed and written to disk immediately when available. As described earlier in this paper, we now employ a two-stage buffering process, where data is flushed from buffers when the buffers are full, rather than on-demand. This improvement brings substantial increases in output bandwidth as well as a more regular and predictable

output time. Figures 7a and 7b as well as Table 1 highlight this dramatic increase in I/O performance.

	Initial	Optimized
Avg. Write Time (s)	10.47	0.30
Std. Deviation (s)	8.36	1.06
Bandwidth (MB/s)	16.04	466.67

Table 1: Comparison between the initial and optimized versions of our *mcw* implementation shows an increase in output bandwidth of 2809%.

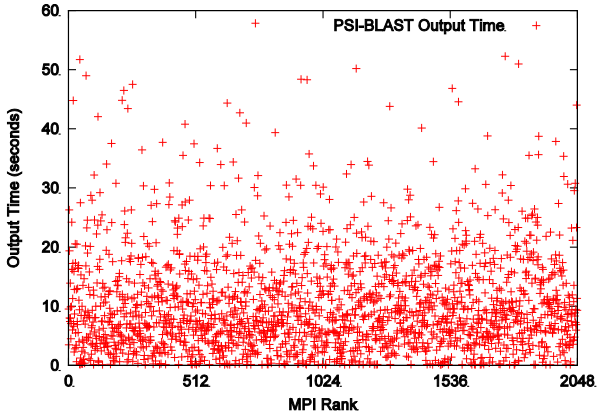

Figure 7a: Total time spent writing data by each of the writer threads in the first *mcw* version running one iteration of PSI-BLAST.

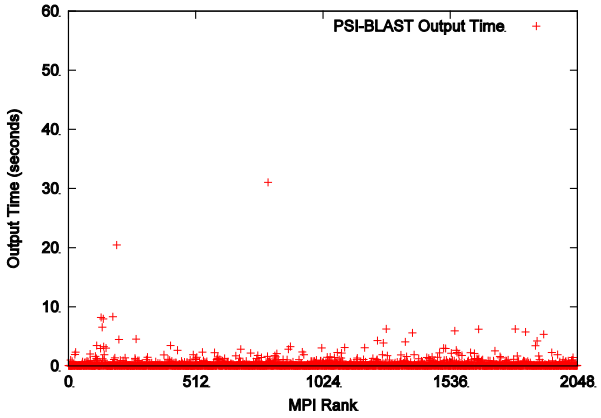

Figure 7b: Time spent writing data by writer threads in the improved *mcw* version. Not only is total output time decreased, but times are more uniform as well.

3 Experimental Results

3.1 Experimental Platform and Datasets

Tests were performed on the National Center for Computational Sciences' supercomputer, Kraken. Kraken is a Cray XT5 machine consisting of 112,896 AMD Opteron compute cores at 2.6 GHz in 9,408 nodes with 147 TB of memory [16]. The sequence database used was the *nr* database containing non-redundant protein sequences from diverse taxa. The *nr* database was obtained from the National Center for Biotechnology Information (NCBI) in April 2011 and contains 13,663,181 sequences consisting of a total of 4,688,826,815 amino acids. When formatted for use with NCBI's PSI-BLAST, the *nr* database totals 11 GB. The query sequences consisted of a carefully selected subset of the database searched against. This subset was selected to be fair in the sense that the selected sequences were of an intermediate length and were of moderate computational complexity.

3.2 Scalability Results

Due to the iterative nature of the PSI-BLAST algorithm there is a large variation in search time among different query sequences, even among sequences with similar lengths. Due to this wide divergence in processing times, we adopt the strong scaling methodology to demonstrate the scalability of PSI-BLAST with our *mcw* wrapper. This is done because the same input sequences will be used for every test, avoiding the issue of wide divergence of processing times that could skew results if utilizing the weak scaling methodology that uses different input query sequences for jobs running with different numbers of cores. Figure 8 shows a number of query sequences – each with a length of 500 amino acids – and their runtimes. This unpredictability of processing time of sequences with PSI-BLAST supports the need for an on-demand work distribution architecture that can actively respond to unpredictability and changes in workload distribution.

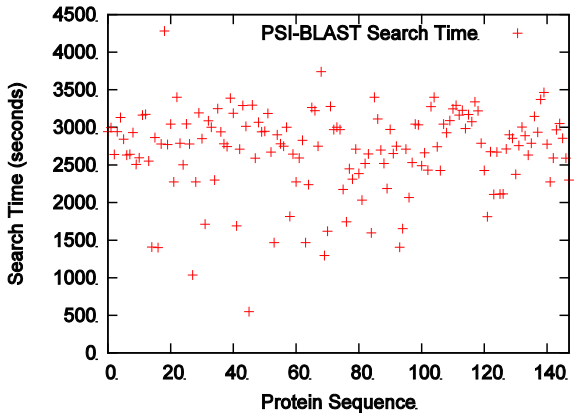

Figure 8: Graph of the search time spent processing PSI-BLAST on selected 150 sequences with 500 AAs from the *nr* dataset. The runtime is not particularly predictable and varies from a minimum of 548 seconds to a maximum of 4,282 seconds – nearly an order of magnitude – in our testing with query sequences of equal length.

Strong Scaling Tests

These tests are done to show how a fixed set of query sequences used with PSI-BLAST perform with differing numbers of cores. This methodology can be seen as a stronger test of scalability, as the same exact input file is used in each test, and the only variable changing between tests is the number of cores used to processes the input data. Results from this testing methodology shows near

linear scalability using HSPp-BLAST which are included in Figure 9.

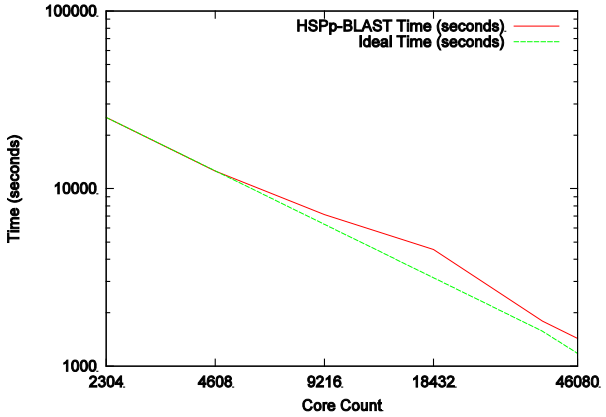

Figure 9: Graph showing scalability of our improved PSI-BLAST on NICS's Kraken utilizing the strong scaling methodology, where each test uses the same query file with 230 million amino acid (million sequences). The red line is derived from experimental runs while the green line represents ideal scaling from 2,304 cores. Currently we can only run 24 hour jobs on Kraken and thus we cannot submit jobs with core count smaller than our smallest job shown above which takes close to 24hours to finish.

4 Conclusion

This paper demonstrates the use of efficient I/O management along with dynamic load balancing that are the key components of scaling PSI-BLAST to tens of thousands of cores which was previously considered computationally prohibitive. By using shared memory, large pages, and buffering techniques we were able to improve the performance of the application on a single core by up to 60% and scale this to tens of thousands of cores, thus improving the overall performance by many orders of magnitude. The results show near linear scalability achieved by this approach while at the same time retaining the original functionality of the NCBI BLAST code. With this approach one can analyze millions of sequences using hundreds of thousands of cores on supercomputers in hours when compared to months or even years of computing on conventional clusters, which can facilitate very large scale genomic data analysis for rapid novel knowledge discovery. We have generated HSP-NCBI module with "blastall" function on Kraken supercomputer which is currently available to all user and will be adding psi-blast function here in future.

5 References

[1] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman, "Basic local alignment search tool," Journal of Molecular Biology 215(3):403-10, 1990.

[2] S.F. Altschul, T.L. Madden, A.A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D.J. Lipman, "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs," Nucleic acid research, Vol 25, No 17, 3389-3402, 1997.

[3] D.W. Mount, Bioinformatics: Sequence and Genome analysis, Second edition 2004.

[4] W.R. Pearson, "Comparison of methods for searching protein sequence databases," Protein science 4:1150-1160, 1995.

[5] PSI-BLAST tutorials. http://www.ncbi.nlm.nih.gov/BLAST/tutorial/Altschul-2.html

[6] A. Darling, L. Carey, and W. Feng, "The design, implementation, and evaluation of mpiBLAST," Proceedings of ClusterWorld, vol. 2003, 2003.

[7] H. Lin, X. Ma, and . P. Ch, "Efficient data access for parallel blast," in International Parallel and Distributed Processing Symposium, 2005.

[8] N. Camp, H. Cofer, and R. Gomperts, "High-throughput BLAST," SGI White Paper, 1998.

[9] E. Chi, E. Shoop, J. Carlis, E. Retzel, and J. Riedl, "Efficiency of shared memory multiprocessors for a genetic sequence similarity search algorithm," 1997.

[10] S.J. Sul and A. Tovchigrechko, "Parallelizing BLAST and SOM algorithms with MapReduce-MPI library," IEEE International Parallel & Distributed Processing Symposium, 481-489, 2011.

[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," IEEE/ACM SC 2008, 1–11, 2008.

[12] S.D. Kahn SD. On the Future of Genomic Data. Science, Vol 331:728-9, 2011

[13] D. Wheeler, "SLOCCount," Available from http://www.dwheeler.com/sloccount

[14] P. Deutsch. Et al., "Zlib compressed data format specification version 3.3." Informational Memo Aladdin Enterprises, 1996.

[15] C. F. S. Inc., "Lustre: A scalable, high performance file system." Technical report, White paper, 2002.

[16] NICS, "Kraken — National Institute for Computational Sciences," Available from http://www.nics.tennessee.edu/computing-resources/kraken .

Acknowledgments

This research used resources at the National Institute for Computational Sciences (NICS) funded by National Science Foundation (NSF) and also supported in part by the NSF grant EPS-0919436.