View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by University of Tennessee, Knoxville: Trace

University of Tennessee, Knoxville

TRACE: Tennessee Research and Creative Exchange

Masters Theses

Graduate School

12-2015

HSP-Wrap: The Design and Evaluation of Reusable Parallelism for a Subclass of Data-Intensive Applications

Paul R. Giblock
University of Tennessee - Knoxville, pgiblock@vols.utk.edu

Follow this and additional works at: https://trace.tennessee.edu/utk_gradthes

Part of the Numerical Analysis and Scientific Computing Commons

Recommended Citation

Giblock, Paul R., "HSP-Wrap: The Design and Evaluation of Reusable Parallelism for a Subclass of Data-Intensive Applications. " Master's Thesis, University of Tennessee, 2015.
https://trace.tennessee.edu/utk_gradthes/3580

This Thesis is brought to you for free and open access by the Graduate School at TRACE: Tennessee Research and Creative Exchange. It has been accepted for inclusion in Masters Theses by an authorized administrator of TRACE: Tennessee Research and Creative Exchange. For more information, please contact trace@utk.edu.

To the Graduate Council:

I am submitting herewith a thesis written by Paul R. Giblock entitled "HSP-Wrap: The Design and Evaluation of Reusable Parallelism for a Subclass of Data-Intensive Applications." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Michael D. Vose, Major Professor

We have read this thesis and recommend its acceptance:

Gregory D. Peterson, Jeremy Holleman

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

HSP-Wrap: The Design and Evaluation of Resusable Parallelism for a Subclass of Data-Intensive Applications

A Thesis Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

Paul R. Giblock

December 2015

Copyright ©2015 by Paul R. Giblock
All rights reserved

Acknowledgements

The research described in this paper used resources at the National Institute for Computational Sciences and Extreme Science and Engineering Discovery Environment (XSEDE). It was developed as part of research supported in part by NSF grants EPS-0919436 and OCI-1053575. I would like to thank Bhanu Rekepalli for his support, Michael Vose for his guidance, and the NICS user assistance group for maintaining a world-class research facility. I would also like to thank Aaron Vose for his role in developing some of the prior work for this project and Eduardo Ponce for contributing to the port of HSP-Wrap to the Beacon platform. Finally, I sincerely thank my friends and family for their support and patience throughout my graduate studies.

Abstract

There is an increasing gap between the rate at which data is generated by scientific and non-scientific fields and the rate at which data can be processed by available computing resources. In this paper, we introduce the fields of Bioinformatics and Cheminformatics; two fields where big data has become a problem due to continuing advances in the technologies that drives these fields: such as gene sequencing and small ligand exploration. We introduce high performance computing as a means to process this growing base of data in order to facilitate knowledge discovery. We enumerate goals of the project including reusability, efficiency, reliability, and scalability. We then describe the implementation of a software scheduler which aims to improve input and output performance of a targeted collection of informatics tools, as well as the profiling and optimization needed to tune the software. We evaluate the performance of the software with a scalability study of the Bioinformatics tools BLAST, HMMER, and MUSCLE; as well as the Cheminformatics tool DOCK6.

Table of Contents

List of Figures

Abbreviations and Symbols

API	application programming interface
BLAST	Basic Local Alignment Search Tool
FIFO	first in, first out
GPU	graphics processing unit
GPGPU	general-purpose computing on graphics processing unit
HPC	high performance computing
HTC	high throughput computing
I/O	input and output
IPC	interprocess communication
MIC	Many Integrated Core
MPI	Message Passing Interface
NCBI	National Center for Biotechnology Information
NICS	National Institute for Computational Sciences
nr	non-redundant (protein database)
NSF	National Science Foundation
OSS	object storage server
OST	object storage target
ORNL	Oak Ridge National Laboratory

PBS	Portable Batch System
SHM	shared memory segment
SIMD	same instruction, multiple data
TLB	translation look-aside buffer
XSEDE	Extreme Science and Engineering Discovery Environment

Chapter 1

Introduction

Our research is the product of an National Science Foundation (NSF) grant to discover methods that promote knowledge discovery through the use of large-scale computational resources. In particular, we are focused on leveraging the high performance computing (HPC) resources available at JICS, a collaboration between Oak Ridge National Laboratory (ORNL) and The University of Tennessee established in 1991. We work with several groups of domain scientists who wish to solve problems that take far too long to complete on traditional computing architectures. These problems fall into the fields of bioinformatics and cheminformatics. The two fields are known for their data intensive nature, and we believe they can benefit from the enormous power of the supercomputers and computing clusters available at The University of Tennessee and JICS. These machines are not only available to us, but they also represent a sample of the HPC resources available in other research centers, broadening the potential application of our results.

The work in this paper stems from research led by Haihang You and Bhanu Rekepalli while optimizing the HMMER software for HPC systems [1]. The optimizations were later generalized by Bhanu and Aaron Vose for scaling the BLAST software [2]. Aaron Vose and I worked on developing this software into the first mature version, MCW, in 2011 [3]. MCW evolved over time and was reimplemented as HSP-Wrap. HSP-Wrap is the focus of the paper, and we describe the iterative approach to designing the software as well as how it benefits the informatics tools we targeted. The solution is benchmarked and we visit the potential shortcomings of HSP-Wrap. Overall, it is a reusable solution and provides promising scalability on many HPC systems.

Chapter 2

Background

In this chapter, we discuss the motivation for this study. In particular, we describe the bioinformatics and cheminformatics software which our work targets. We introduce the growing demand of data processing, why large-scale data processing and analysis is needed, and how high speed computing can meet this demand. We then describe the fundamental requirements and goals of our work.

2.1 Data explosion

The data generated in many scientific and non-scientific fields is growing rapidly. Science has seen an ever-growing rate of data generation due to projects such as the Large Hadron Collider which contains approximately 150 million sensors, or scientific simulations such as those used to predict climate change. The biotechnology and pharmacology fields contribute massive amounts of molecular, protein, and nucleotide data. For instance, advances in the decoding of the human genome has led to the development of DNA sequencing machines that are orders of magnitude faster and 10 000 times less expensive than they were only 10 years ago [4]. This trend in data growth is not limited to science however. The computerization of many systems such as phone call records, surveillance, weather sensors, medical records, and commerce generate large amounts of data. The Internet contributes data such as documents, search history, email, user tracking, and data generated through social networks.

2.2 Case-study

We focus on the dramatic growth of data in life sciences which has surpassed performance improvements of the computing systems used to analyze such data [5]. In particular, we

Table 2.1: FASTA character representation, C, of nucleotide residues found in DNA, RNA, and a selection of amino acids residues found in proteins

C	Nucleic Base
A	Adenine
C	Cytosine
G	Guanine
T	Thymine
–	Gap

(a) DNA sequence

C	Nucleic Base
A	Adenine
C	Cytosine
G	Guanine
U	Uracil
–	Gap

(b) RNA sequence

C	Amino Acid
A	Alanine
B	Asparagine or Aspartic Acid
C	Cysteine
D	Aspartic Acid
...	*22 others*
X	Any
–	Gap

(c) Protein sequence

are interested in two areas of information processing that have a substantial impact on the quality of human life; these are bioinformatics and cheminformatics. Our goal is to improve a collection of tools used by researchers in these fields. The rest of this section focuses on the bioinformatics and cheminformatics fields as well as the programs that we targeted as candidates for optimization.

2.2.1 Bioinformatics software

Bioinformatics is the area of study involved with the retrieval, storage, and analysis of biological data. This data includes amino acid sequences (proteins), nucleotide sequences (DNA and RNA), protein domains, protein structures, and entire genomes [6]. Bioinformatics grew out of the tremendous amount of data generated by biologists, and the need to store and analyze it. Processes of interest to biologists include: finding or aligning similarities among different sequences, finding patterns within a biological sequence, building phylogenic trees to model ancestry, clustering data based on existing groups, and more [7]. A sequence is an ordering of amino acid or nucleic acid residues as found in a single molecule of a protein or nucleotide, respectively, and can be modeled as a string of these residues.

Computer systems can be used to meet the needs of bioinformatics due to their storage and processing capacities. Bioinformaticians, computational biologists, and computer scientists are typically involved in designing and implementing these specialized software tools. We will briefly describe some readily available data standards and bioinformatics software to introduce the requirements of the software as well as how the tools are used in practice.

A commonly used format for sequence data interchange is the FASTA file format; originally designed for the FASTP/FASTA software described by Lipman and Pearson in 1985 [8]. The format is basically a string of characters where each character represents a single residue.

For instance, table 2.1 shows some of the codes seen in DNA, RNA, and Protein sequences. A FASTA sequence is prepended with a description which is free-form text, but is typically used to store identifying information about the sequence such as numerical identifiers and the scientific name of the molecule.

NCBI BLAST

NCBI BLAST is an implementation of the Basic Local Alignment Search Tool (BLAST) published in 1990 by the National Center for Biotechnology Information [9], and is one of the most widely-used tools for sequence similarity searches. BLAST can perform comparisons between sequences and a sequence database by finding local regions of similarity between the two sequences. There are variations of the algorithm for different research needs, the most popular are:

Protein BLAST Compares a protein query sequence against a protein database.

Nucleotide BLAST Compares a nucleotide sequence against a nucleotide sequence database.

Translated BLAST Compares a nucleotide sequence against a protein database, or vice versa. This is accomplished by converting the nucleotide sequences to different possible protein sequences in a process known as framing.

PSI-BLAST Compares a protein sequence against a protein, but searches the database using a scoring matrix which is updated through multiple iterations of the search [10]. The matrix contains scores for each possible amino acid at each specific position of the query sequence. Hence the name: Position-Specific Iterated BLAST. PSI-BLAST is more sensitive than the other mentioned algorithms in finding weak, yet significant, similarities between sequences.

The general structure for all of these variants remains the same, even though the kernel changes. The program works by opening a sequence database from disk, performing whichever operations are needed to initialize the algorithm, then sequentially reading an input FASTA file for use as query sequences. The results of each search are written to an output file upon completion.

HMMER

HMMER by Howard Hughes Medical Institute's Janelia Farm is another method for performing similarity searches of a sequence database [11]. HMMER can also be used for protein domain identification, which is used to predict biological function. Moreover, it can also perform remote homolog searches due to the mathematical model in use. Unlike BLAST, which uses a pairwise method, HMMER uses Hidden Markov Models (HMMs) to compute profiles for each sequence. A profile is calculated from multiple alignments of that sequence. HMMER operates similarly to BLAST in structure; database files are opened at first and input is a FASTA file.

MUSCLE

MUSCLE is a tool commonly used for bioinformatics research [12]. MUSCLE creates multiple sequence alignments among a set of biological sequences, which are important in many applications; including crucial residue identification, phylogenetic tree generation, and structure prediction. It outperforms most other multiple sequence alignment algorithms, such as CLUSTALW [13]. MUSCLE operates differently than BLAST and HMMER as there is no database. Instead, MUSCLE accepts a single FASTA file containing multiple sequences. It attempts to identify areas of similarity shared among multiple sequences by arranging the residues of the sequences such that conserved (similar) regions are aligned. This can be visualized by strategically placing gaps between certain residues.

2.2.2 Cheminformatics software

Cheminformatics (less commonly referred to as **chemo**informatics) is the application of informatics techniques applied to chemistry instead of biology. The techniques are primarily used in the process of drug discovery. The cheminformatics field faces many of the same problems bioinformatics has due to data growth. The growth in cheminformatics occurs in the space of molecules and chemical compounds. This includes organic and inorganic compounds. The chemical space grows exponentially when more complex molecules, which have more atoms, are discovered [14]. Cheminformatics is concerned with the storage of data and the search of knowledge that can be discovered in regard to compounds.

An important objective of the field is the creation of large chemical databases. These databases can contain chemical structures where each molecule is represented as a set of atoms with the bonds represented as graph edges. Other structure databases, such as protein databases, represent the proteins as amino acid sequences. In either case, the database may store annotations for each compound including the three-dimensional structure, atomic connectivity, and other traits outside the scope of this discussion [15].

Drug discovery is a critical but complex and costly endeavor. Since small molecule chemical space is currently in the millions and is growing rapidly, performing physical screening is complex and cost-prohibitive even for small subsets of molecules without significant investment in expertise and infrastructure. The technique of molecular docking is used to perform virtual screening. The process involves predicting possible shapes of the target protein and determining if a small compound (a ligand) is able to attach to any locations within the protein. This provides an indication as to how the compound may act as an inhibitor of the protein.

UCSF DOCK

DOCK 6 was developed at the University of California, San Francisco and is an application that identifies the low energy binding modes of small molecules with proteins [16, 17]. Ligand docking is efficiently accomplished by randomly rotating the bonds of the ligand in the protein's binding pockets and calculating the interactions by geometric matching (Van der Waals) and force-field scoring (electrostatics). DOCK 6 scores provide a rapid estimation of approximate binding energies. In order to use DOCK 6, the operator must prepare a configuration file that specifies the target protein, an input file (in MOL2 format) containing a list of molecules to dock, and a set of configuration variables that modify the behavior of the algorithm. DOCK 6 supports writing to multiple output streams: one for logging the progress and scores, and another for writing the transformed molecules once the docking has succeeded.

2.3 High performance computing

High performance computing (HPC) does not have a rigid definition but generally refers to the practice of aggregating computer resources in a way that provides significantly higher performance than one could achieve from a single computing device, such as a desktop or workstation. What is considered "high performance" changes with time. The Cray-1, a vector computer released in 1976 by Seymour Cray would have been considered a High Performance Computer in that age. In fact, that system was considered to be a supercomputer, or a computer on the pioneering edge of computational capacity. The design and capacity of supercomputers have evolved over time. Early supercomputers employed only a few processors which allowed them to retain a familiar shared memory architecture [18]. Top supercomputers in the mid 1990s instead had thousands of processors [19]; however, shared memory architecture is difficult to scale due to increased contention for memory bandwidth and the cost of cache coherence.

Therefore, the trend has been toward distributed memory systems; where each processor has its own private memory, and communication between processors is performed by message passing. Organizations who once could not afford to build a smaller-scale HPC system can now connect desktop computers or specially built nodes together into a cluster to run distributed programs. Leading supercomputers differentiate themselves by optimizing the speed of interconnects and choosing topologies with higher connectivity such as a two or

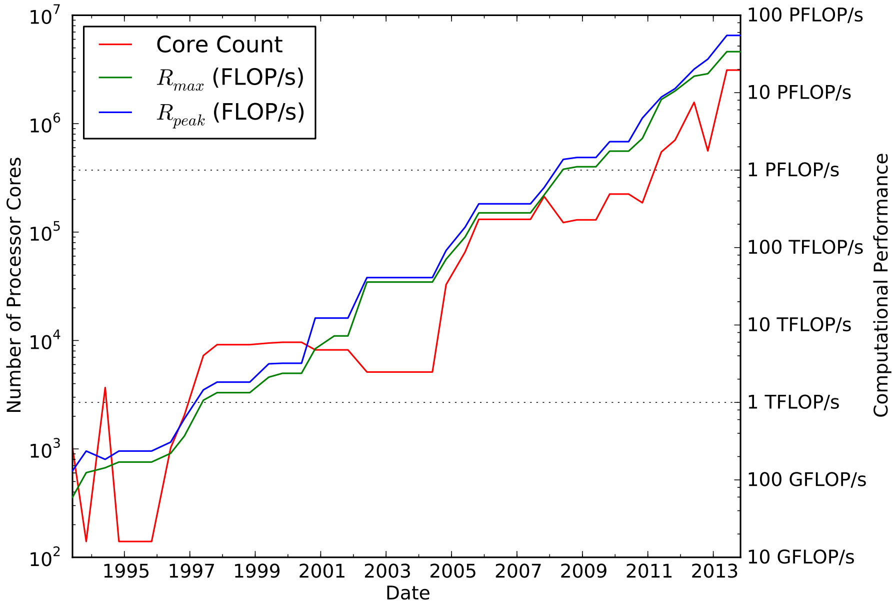


Figure 2.1: Number 1 ranked supercomputer speed and core counts over time

three-dimensional torus [20, 21]. They must also handle the heat produced by such a massive collection of processor cores. Figure 2.1 shows the core count and speed of the fastest supercomputer in the world over the course of nearly twenty years as recorded by TOP500. We can see that the computational capacity has increased exponentially over the years, but not directly due to increased core counts.

2.3.1 Target architectures

Supercomputers are applicable to many tasks including weather forecasting [22], cryptanalysis [23], and of course computational science. The presence of the Extreme Science and Engineering Discovery Environment (XSEDE), a community-driven project to enhance scientists' access to high-end computing resources [24], can be viewed as an indicator of this demand. Thus, our goal is to make the informatics applications described in section 2.2 available on HPC resources. In particular, we will focus on scaling the applications of interest to run on machines available at NICS, The National Institute for Computation Sciences, and the University of Tennessee, Knoxville. Not only are these machines accessible to us for application development and benchmarking, but they represent the general architecture seen in the vast majority of supercomputer and cluster deployments.

Kraken and similar architectures

Kraken was a Cray XT5 machine consisting of 112 896 AMD Opteron compute cores at 2.6 GHz in 9408 nodes with 147 TB of memory [25]. The nodes of Kraken each had 16 GB of system memory and two six-core AMD Opteron processors. The nodes were connected with the SeaStar Interconnect, which features a 3-D torus topology. Kraken achieved a maximum performance of 1.17 PFLOP/s, and it was the world's third fastest supercomputer in 2009 as ranked by TOP500 [26]. We choose to study performance on this machine because it is representative of many homogeneous processor, distributed memory architectures; and the high core count makes it fit for scaling studies. Kraken was decommissioned on April 30, 2014.

We also consider the Newton HPC Program at The University of Tennessee, Knoxville; developed to facilitate research at this and collaborating institutions [27]. The system has over 4700 x86-64 processor cores, over 9 TB of memory, and is divided into multiple heterogeneous processor, distributed memory clusters. The cluster configurations vary in CPU model, core count, memory size, and interconnects. Newton is useful for testing the performance on a less massive machine, but it is also valuable for debugging and preliminary scalability testing before committing time from our limited allocation on Kraken.

Heterogeneous computing

Combining different types of processors into a single system is not anything new. Long have computing systems contained specialized ASICs, math processors, and field programmable gate arrays (FPGAs). Heterogeneous computing in the HPC world, however, has been less common until more recently. That is, until the rise of graphics processing units (GPUs) in late 1990s and early 2000s and general-purpose programming on GPUs (GPGPU) . The GPGPU technique allows the programmer to use the many underlying SIMD multiprocessors of a GPU for accelerating certain algorithms such as those which require running the same procedure over independent data. This technique really skyrocketed when CUDA, a programming interface for GPUs, was released by NVIDIA in 2007; as well as the OpenCL and OpenACC standards which followed afterwards.

Intel announced the Many Integrated Core Architecture (Intel MIC) in 2010, and is now available in their Xeon Phi line of products. Intel MIC is based on the x86 architecture, but a processor currently contains somewhere between 50 to 100 cores; several times more than a conventional processor. The Xeon Phi is packaged as a co-processor intended to offload expensive calculations away from the system's main processor. The x86 cores are relatively low power at about 1.2 GHz, but they do feature 512-bit SIMD units, and an ultra-wide ring bus connecting the cores and memory [28]. The Xeon Phi co-processors can be programmed using standard libraries such as OpenMP and OpenCL as opposed to a language such as CUDA. Additionally, the co-processors run an embedded Linux system that allows software to be uploaded to the card and executed in a POSIX environment. Libraries such as MPI can be used to coordinate computation between the main processor and the co-processors in this mode.

We use the Beacon Cluster at NICS for studying the Intel MIC architecture. This system was funded by the NSF with the purpose of optimizing scientific programs for the co-processors. The system features 48 compute nodes, each with 2 8-core 2.6 GHz Intel Xeon E5-2670 processors, 256 GB of memory, and 4 Intel Xeon Phi 5110P co-processors. The 5110P co-processors each feature 8 GB of memory and 60 cores. This brings the total for Beacon to 768 Xeon cores, 11 520 MIC cores, 12 TB of system memory, and 1.5 TB of co-processor memory. The combined performance of the conventional processors and co-processors is 210 TFLOP/s [29, 30]. We experiment with Beacon to determine the performance benefit of the co-processors versus code that does not utilize the co-processors at all.

Distributed file systems

One aspect of the target systems described above that we have not discussed is the persistent file system. Kraken, Beacon, and Newton all use the Lustre distributed parallel file system. A distributed file system is a network file system where the data resides across multiple network hosts. The advantages of a distributed file system over a single-server network file share are high performance and high availability. Lustre works by maintaining a metadata server (MDS) and multiple object storage servers (OSS). The MDS stores information such as filenames, directory hierarchy, and file permissions on a metadata target (MDT). An object storage server (OSS) provides access to the file contents stored on one or more object storage targets (OST). These targets are simply volumes and can be a hard drive, RAID volume, or potentially another storage medium. File accesses are handled by the client by first querying for the file on the MDS. File metadata, including the logical layout of the file is returned. The logical layout allows the client to locate the objects which contain the file contents on one or more object storage targets [31]. Other distributed file systems, such as HDST (Hadoop Distributed Filesystem) and MooseFS may work differently, but the facts remain that the file system is remote and the resource is shared. Therefore, observation of access patterns, and attempting to mitigate performance penalties is a focus area of this paper.

2.4 Prior work

There are many parallel versions of some of the informatics tools listed in section 2.2. For instance, mpiBLAST [32] and pioBLAST [33] achieve scalability on high performance computers through the use of techniques such as database or query sequence partitioning. However some of the BLAST functions, such as PSI-BLAST and PHI-BLAST, are not available in these optimized versions due to the iterative nature of PSI-BLAST and the complexities in parallelizing the algorithm [3]. MPI-HMMER is a popular scalable version of the HMMER suite that employs database fragmentation and double-buffering for speedup. However, MPI-HMMER does not scale well past approximately 512 processes as described by researchers at JICS [34]. DOCK6 contains built-in support for MPI to allow the code to be executed on a cluster. However, there are no additional optimizations beyond the distribution of work. There are optimized versions that focus on porting the routines to run on GPU or FPGA devices, while others focus on particular supercomputer architectures.

Researchers from JICS (Rekepalli, Halloy, and Zhulin) created the first version of HSP-HMMER in order to overcome the limitations of MPI-HMMER, and to scale the software on the Jaguar supercomputer [34]. HSP-HMMER performs much better due to the efficient job scheduler, but is limited to the `hmmscan` routine. This work was later extended with a hierarchical dynamic load balancer and I/O optimizations such as database broadcasting and output buffering [1]. The techniques used in this system form the foundation of our work. The concept was then applied to BLAST [2], but with an I/O "wrapper library" to abstract the optimizations from the main Protein and Nucleotide BLAST routines.

We took the wrapper design, generalized it further, and integrated it with PSI-BLAST in order to finally run PSI-BLAST efficiently on Kraken [3]. We chose to focus our efforts on the development of this reusable wrapper, MCW, due to several reasons. Although parallel versions of many of the tools used in the case study already exist, we realized shortcomings in many of them such as missing routines, sub-par performance for large node counts, or a reliance on a particular architecture. We hoped that focusing on the wrapper would allow for our existing and future optimizations to be orthogonally applied to all of our tools of interest. Furthermore, a resusable solution can be used for the development of new tools. More information on the motivation for reusability is in section 3.3.

Other software libraries provide enhanced I/O routines, but they do not automatically balance work across the processors. These libraries include MPI-IO, SIONLib [35], and ADIOS [36]. Additionally, these tools tend to focus on collective operations on rigidly structured data formats such as HDF5 and NetCDF which is a poor fit for the informatics tools' varying input and output sizes and execution times. ADIOS could possibly be used in the future to support an additional subset of tools, with the wrapper providing load balancing. The IOFSL scalable I/O forwarding library is possibly the closest to our needs in that it exposes scalable general purpose I/O routines. Again, there is no load balancer or any type of task scheduling, although file accesses themselves can be “wrapped”.

Chapter 3

Methods

3.1 Design goals

Previous chapters have presented the background of the software, the basics of the target computing architectures, and our primary goal of running this software on these computers. We will now enumerate more fine-grained goals and requirements. The NSF created a program in 2009 to fund projects that boost the ability for people to build and use data-intensive computing systems [37]. They focus on a number of approaches such as parallel algorithms, parallel programming, programming abstractions, and determining software where the research could be applied. They also outline some basic goals, which are to provide reliability, efficiency, availability, and scalability. We will also adopt these goals as our own within the context of the informatics software we reviewed earlier. This leaves us with the requirements:

Provide parallelism across nodes and cores. We attempt to keep the processors as saturated as possible. Time spent not computing is inefficient and negatively affects scalability.

Scalable input and output routines. I/O should reduce negative performance impacts of the network interconnects and the parallel distributed file system. I/O should be efficient, limiting needless copying. Time spent waiting on I/O impacts scalability. We also foresee possible availability issues should the software overwhelm the file system with data requests.

Fault-tolerant. The software should be resilient against failure. It would be ideal if the solution can continue functioning after failure, if infrastructure permits (fail softly). Alternatively, the system should be resumable to avoid expensive re-computation.

Extendable for workflows. Many informatics tasks can be modeled as a workflow. That is, a set of processing stages that read input from the previous stage and forward the results to the next stage. The design should be adaptable to workflows as a future extension.

Easily Reusable. The system should be readily available for use in new or existing software. C and C++ will be targeted first, however it should be extendable to other languages or environments such as Java or Python.

A survey of the target architecture reveals that the following assumptions can often be made with regard to the HPC system and available facilities; simplifying our research:

POSIX compliance or at least near-compliance. The systems we use are all based off of Linux; even the hosted environment of the Intel Xeon Phi. This is largely due to the openness of the platform which allows the hardware vendors to tailor the operating system to the hardware. The use of free (as in beer) software greatly reduces the cost of the system due to reduced software licensing; especially when multiplied by the number of compute nodes in service.

Portable Batch System or PBS. The systems we have seen use a PBS system for submitting work (jobs) to the compute nodes. This is a time-sharing scheduler that maintains a queue of submitted jobs and executes a job when the resources become available. We will assume a PBS system such as OpenPBS, SGE, or TORQUE is in use. If not, then the user must be able to concurrently start the processes on many nodes by some other means.

MPI or Message Passing Interface, is a standardized API for passing messages between multiple processes on multiple nodes. The specification is popular, and many HPC deployments feature fine-tuned implementations that take advantage of the system's interconnects and characteristics. We use MPI as the basis for all inter-node communication.

3.2 Profiling of original code

We started by looking at BLAST, PSI-BLAST in particular. What we realized through code inspection is that the process reads several database files, memory maps many of them, and then begins to iterate over input sequences from the user specified FASTA file. We were immediately concerned by this access pattern when scaled to many processes. A naive implementation would result in many requests for the same data from the file system. We implemented this naive solution in order to establish a baseline for the scaling study. The

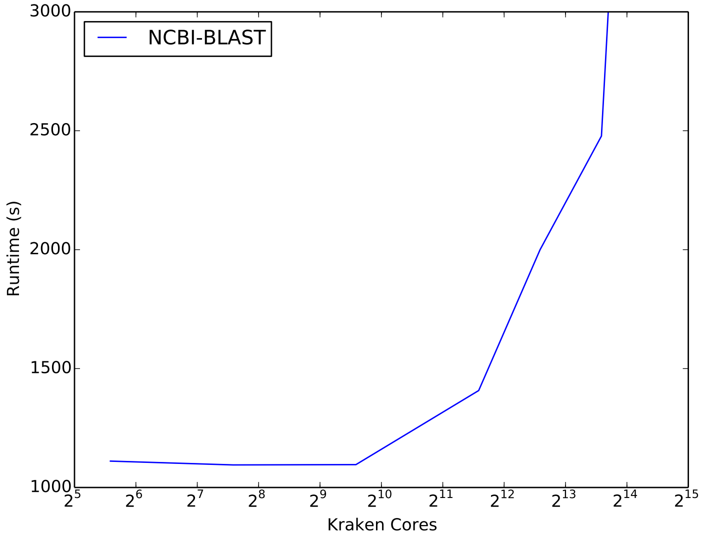


Figure 3.1: Weak scaling of the naive BLAST implementation. I/O contention causes performance degradation.

implementation spawns one blast process for each compute core and runs several sequences on each core. Each core runs the same set of sequences. Figure 3.1 shows the weak scaling characteristics as the number of nodes is increased while the work assigned to each node remains the same. This is in contrast to a strong scaling study where the total problem size is fixed.

An ideal linear scaling would appear as a nearly horizontal curve with the runtime remaining constant. The figure instead shows that scalability begins to suffer after approximately 1000 cores, and becomes increasingly worse after that. There must be some influence external to the actual BLAST kernel since the jobs executed by each core are identical. We grew suspicious of the I/O. The degradation of I/O throughput was verified by using the CrayPat [38] profiling tool and manually injected timing code. Therefore, improved input and output behavior became our first priority. We followed the guidelines set by a number of NICS researchers to aide in optimizing I/O, namely: limit the impact of latency by performing I/O in fewer, larger chunks; limit overhead by reducing the number of metadata changes; favor writing data contiguously; and avoid file system contention [39].

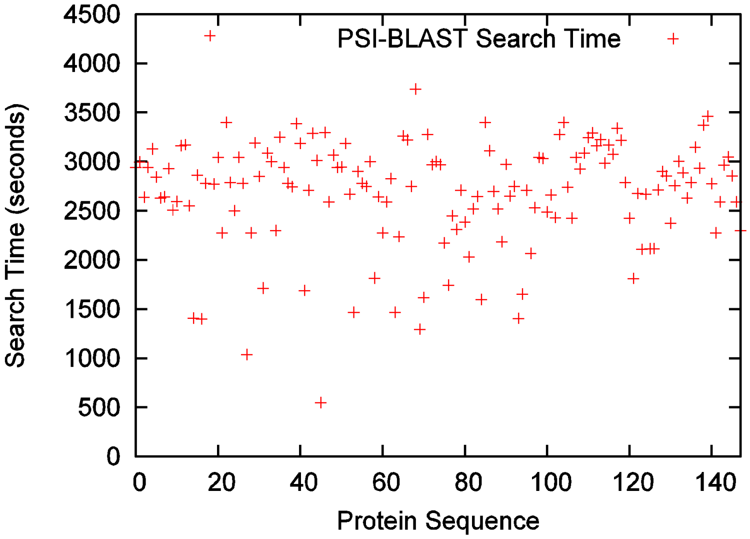


Figure 3.2: BLAST runtimes vary greatly for 150 proteins of the same length.

3.2.1 Dynamic load balancing

The previously mentioned study processes the same set of input data for each compute core. This operation is not useful in practice as it results in duplication of each computation. Instead, we wish to distribute one set of input data across all of the compute cores. In doing so, a speedup should result by virtue of the division of labor. There are several approaches we could take, and they can be classified as either static or dynamic load balancing. Static load balancing would split the input set into n sets; one for each of the n cores. Dynamic load balancing distributes the input on demand as the program runs; there is no predetermined partitioning of the input as seen in static load balancing.

The choice between a static or dynamic approach depends on the runtime characteristics across different inputs. A static approach would be sufficient if the inputs all run in an identical amount of time, or if there is a way to predict the runtime from the input itself. We ran PSI-BLAST on 150 different sequences selected from the nr dataset, each with 500 amino acids. Figure 3.2 shows that the runtime varies from a minimum of 548 seconds to a maximum of 4282 seconds — nearly a factor of ten. This preliminary profiling suggests we should use a dynamic load balancing technique. Moreover, even if BLAST times could be determined statically, we would possibly want dynamic load balancing as it is more robust. A dynamic solution could more easily reschedule work from a failed compute node, or handle heterogeneous tasks.

Table 3.1: Source lines of code and primary language of implementation for the informatics tools as reported by `sloccount`.

Tool	BLAST	HMMER	MUSCLE	DOCK6	**Total**
Primary language	C	C	C	C++	–
Lines of code	1 300 000	35 000	28 000	147 000	1 510 000

3.3 The need for reusability

We propose a solution that facilitates scaling of the targeted informatics software described in section 2.2 on the architectures described in section 2.3.1. There is a common theme that exists in the software we consider. They work by optionally loading a database. Then, they run some algorithm sequentially across an ordered set of inputs. The results of the algorithms are written to one or more output files during or at the end of the algorithm's execution. Accordingly, we hope to find common bottlenecks in how the input and output data is handled, and then try to enhance the performance by providing optimizations for these routines or patterns.

A reusable solution would be ideal. The size of the targeted informatics software is large and we do not want to create an individualized solution for each of the tools. Table 3.1 shows the number of source lines of code for the software rounded to the nearest thousand, as well as which language is used for the majority of the code. The figures were calculated by the `sloccount` tool by David Wheeler [40], and as such, the numbers are not 100% accurate. Our solution must be trusted to produce accurate and correct results. To this end, it is desirable to make as few changes to the original code as possible in order to avoid introducing adverse changes. Maintaining forked software is also expensive due to the cost of merging and testing changes from the upstream developers. This suggests a wrapper strategy.

3.4 Isolate program I/O

We must identify which program routines lead to troublesome I/O access patterns if we wish to optimize them. In this section we describe how this challenge forced us to evolve our approach over the course of the project. We are only concerned with how to isolate and override routines from the programs, as opposed to also optimizing the underlying low level facilities utilized. The override functionality is implemented in the HSP-Wrap scheduler/wrapper which is described in detail in section 3.5. This scheduler is accessed by the program through interprocess communication (IPC).

3.4.1 Ad-hoc approach

Our first prototypes of a scaled PSI-BLAST used an ad-hoc approach. We inspected the source code to determine which routines are responsible for the input and output of the program. The routines of interest are the loading of the database, reading of the input file, and writing of results. We re-implemented these functions to instead retrieve data from the scheduler and locally cached data. This was simple, and it helped to prove the viability of our technique. The need for a more general approach was evident as the coding effort progressed. We decided to look for a better way to alter the software before expanding to other informatics tools.

3.4.2 Library of standard I/O routines

The standard way to reuse routines is by creating a module, or library. We could define our own application programming interface (API) to expose optimized I/O routines to the informatics software, but we decided that this is not necessary as there are already standard APIs for us to model the library after. These APIs are contained within the standard libraries of C and C++. This decision eliminates the effort of designing a new interface, and eases the effort of porting the informatics software since the new library can be interposed in place of the routines from the standard libraries. The effort produced two libraries, `libstdiowrap` and `libstdiowrap++`; which emulate the C `stdio` and C++ `std::fstream` interfaces, respectively.

C library (`libstdiowrap`)

The `libstdiowrap` library is responsible for redirecting input and output file streams from C programs to the underlying HSP-Wrap wrapper (which is described in detail in section 3.5). The library provides equivalents to 20 of the C standard library functions available in `stdio.h`; including opening files (`fopen`), reading (`fread`, `fgetc`, etc), and writing (`fputc`, `fprintf`, etc). In addition, we provide replacements for the POSIX `stat`, `open`, `close`, and `mmap` functions; although `open` and `close` are only included to support memory mapping. We call these replacement functions "wrapped", and they are named by prepending the original function name with "`stdiowrap_`". So, for example, `int fclose(FILE *)` becomes `int stdiowrap_fclose(FILE *)`.

A program written in C can be easily modified to take advantage of our I/O optimizations and scheduling by prepending "`stdiowrap_`" before relevant function calls. We found that some programs should have all of the I/O functions wrapped. We created convenience C preprocessor macros to assist with this use case:

Listing 3.1: Automatic function wrapping through macros

```
#ifdef STDIOWRAP_AUTO
#    define fopen(a,b)       stdiowrap_fopen((a),(b))
#    define fclose(a)        stdiowrap_fclose((a))
#    define fseek(a,b,c)     stdiowrap_fseek((a),(b),(c))
//    ... snip ...
#endif
```

A program author needs only to define `STDIOWRAP_AUTO` in order to automatically wrap the functions supported by `libstdiowrap`. In our experience, we can completely avoid modifying the source code of some programs by modifying the Makefile so that `CFLAGS` features "`-include stdiowrap.h -DSTDIOWRAP_AUTO`" and by adding "`-lstdiowrap`" to `LDFLAGS`.

`libstdiowrap` emulates the behavior of the standard C routines. The `FILE *` handles are references into `libstdiowrap`'s internal file table. Behind the scenes, `libstdiowrap` is implemented on top of interprocess communication with the HSP-Wrap process, which is used to service I/O requests rather than the OS's system calls. The details of the IPC scheme is expounded in section 3.5.1.

C++ library (`libstdiowrap++`)

We wrote the `libstdiowrap++` library to aide in wrapping C++ programs, DOCK6 in particular. This library is implemented on top `libstdiowrap`. The bulk of the binding is in a custom `std::filebuf` subclass named `stdiowrap::filebuf`. This is an adapter class that encapsulates the `libstdiowrap FILE *` handle and uses `libstdiowrap` functions to implement a C++ filebuf, the basis of C++'s file I/O library. We provide `fstream`, `ifstream`, and `ofstream` classes in the `stdiowrap` namespace for convenience. These classes permit the C++ programmer to use our wrapped routines by replacing select uses of the `std` namespace with `stdiowrap`. For instance, `stdiowrap::ifstream` could be used in place of `std::ifstream` for an input file.

We hoped to find a method to automatically interpose functions similar to the macros used by `libstdiowrap`, but we were unable to discover a satisfactory approach. The main problem is due to C++'s support of namespaces which render macro expansions ineffective. For instance, the `std::fstream` class could be used as `fstream` if the code previously introduced a namespace into the declarative region through the `using` keyword. The class could be known as `fubar::fstream` if the `std` namespace was aliased, or worse: all of these typenames could be valid in different regions of the same source file. Then there is the `typedef` keyword to further complicate things.

The solution is capable enough in practice, even with this limitation. Not all streams needed to be wrapped when we ported DOCK6, so a fully automatic method would not have been useful to us. Also, the programmer only needs to replace the type of variable declarations from the `std` namespace to the `stdiowrap` namespace. There is no need to modify every function call since the polymorphism functionality of C++ satisfies this automatically.

3.4.3 Preloaded runtime library

All of the approaches discussed thus far require changes to the source code, albeit small changes. We experimented with providing a solution which requires no changes to the code. In fact, it requires no recompilation at all. The approach is to use the `LD_PRELOAD` environment variable recognized by at least the Linux program loader. The user is able to list shared libraries in this environment variable. Symbols will be resolved against these libraries before any of the normally referenced shared libraries. This allows for runtime function interposition. The idea is that we can reimplement a number of functions provided by or used by the C runtime library, and our functions will be invoked instead of the standard functions. The ugly truth is that this gets very complicated quickly. The GNU C runtime library (`glibc`) uses layers upon layers of macros to implement the functions we care about. Once we find the function responsible for, say, opening a file; we notice that the function prototype is very different. Internal data structures are used, concerns over corrupting state begin to surface, and the solution would be fragile as it only targets the GNU C runtime — well — some versions of it. We already had BLAST, DOCK6, and HMMER wrapped at this point, so the cost was not worth the endeavor. Also, `LD_PRELOAD` may not be enabled on some systems for security or compatibility reasons.

3.5 The scheduler (HSP-Wrap)

In this section, we describe HSP-Wrap, our flagship scheduler. HSP-Wrap is implemented in C and uses MPI to communicate across nodes. A simplified schematic of the HSP-Wrap architecture is illustrated in figure 3.3. Its primary purpose is to efficiently distribute data to the tool processes, schedule tasks, manage system resources, and to quickly write resulting data.

Files are divided into three classes depending on how they are used by the wrapped tool:

Input. These are the input streams; for example, FASTA files or MOL2 files. These files contain the input that needs to be dynamically load balanced across compute cores. These files behave much like a pipe opened for reading from the application's point of view. The file is not seekable, and can only be read from; although functions such as `ungetc` continue to function. The application does not know how long the file is, but it knows if there is more data available or if the end of the file has been reached.

Output. Output files are analogous with input files, but where the file is written to instead of read from. Again, the file cursor cannot be positioned, and data can only be appended to the end of this type of file stream.

Shared. A shared file is a special type of input file reserved for data shared between instances of the wrapped program; for example, database and configuration files. Shared files act more like a traditional read-only file than a stream. The wrapped application can seek the file position of, and even memory map the shared file. This mechanism also helps to reduce the memory footprint of the distributed application since the typically large database files are shared between processes of a compute node.

One compute node is designated as the master, the remaining nodes are workers. The master node is responsible for loading file data from storage and distributing it to the worker nodes. Each worker maintains a pool of wrapped tool processes in conjunction with a thread that writes output to storage. The master broadcasts the executable and shared file data to all of the worker nodes. Each worker places the received executable files in a working directory dedicated to the node, and copies the shared file data into shared memory segments. Following that, the worker nodes start the tool processes and request the initial input queries from the master; enough for each processor core and some extra to populate a local queue, if available.

HSP-Wrap and `libstdiowrap` service the tool's I/O requests as the processes run. Shared files are stored in shared memory segments (SHMs), and access routines are fairly simple since the data is read only and entirely resident in memory. Reading from an input file employs a hierarchical approach. `libstdiowrap` services read requests by accessing a shared memory segment dedicated to the process. The buffer typically contains a single input record; for instance, a single FASTA sequence. `libstdiowrap`, upon reaching the end of the buffer, requests another input from HSP-Wrap and blocks the process. HSP-Wrap attempts to service the request locally if there are any inputs in the worker node's input queue for that file. If so, the input is copied into the process's buffer and HSP-Wrap signals the process to continue. If the queue is empty or nearly empty, then HSP-Wrap must request more inputs from the master node. The master merely iterates through the input file and sends batches of inputs to the workers as more work is requested. Output is handled without intervention from the master node. Each tool process is assigned SHMs for each output file, and `libstdiowrap` writes output to the SHM. `libstdiowrap` signals to HSP-Wrap when the SHM is full. HSP-Wrap copies the buffer contents to the worker node's output queue, then signals back to `libstdiowrap` to let the process continue. The worker's writer thread periodically flushes data from the output queue to a file in the node's working directory.

Job termination is fairly straightforward. HSP-Wrap responds to each worker's request for

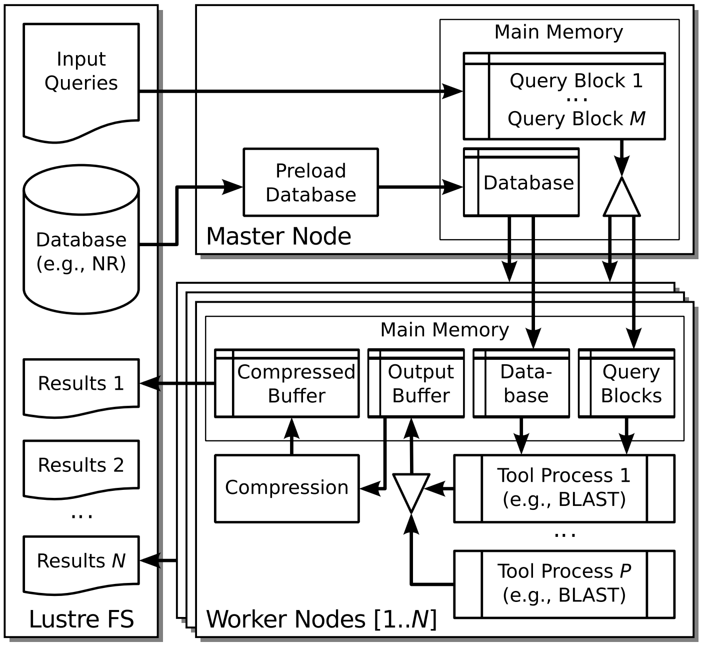


Figure 3.3: The architecture used by HSP-Wrap

more work with an exit response (`WU_TYPE_EXIT`) when there is no more input. The worker node is now in "exiting" mode. It no longer requests input from the master, and instead waits for all of the node's tool processes to end. `libstdiowrap` receives an "end of data" status (`PS_EOD`) from HSP-Wrap when attempting to read more data from an empty input queue while in "exiting" mode. The `libstdiowrap` functions return the expected value for a file handle that has reached its end: `fread` returns a short object count, `fgetc` returns `EOF`, `feof` returns non-zero, etc. The wrapped tool sees this, and begins to terminate the process. The master process is finalized once each worker node receives an "exit" response, and the workers are finalized once each tool process exits and any remaining output is flushed to storage.

3.5.1 Inter-process communication

The IPC scheme used in HSP-Wrap is the result of iterative development. The design required modification as we discovered restrictions of the target HPC platforms in order to maintain portability and high performance. An ideal way to abstract the I/O would be as a virtual file system. This could be accomplished as a kernel module, as a FUSE user-space file system [41], or as any other mechanism supported and enabled by the operating system. This support was unfortunately not available on any of our target platforms. The idea of runtime library interposition noted in section 3.4.3 also failed. This is unfortunate since either of these methods could have superseded the need for `libstdiowrap` and tool recompilation altogether.

Another idea was to use files and named pipes (FIFOs) stored on an actual file system. The files could be filled with shared data while the named pipes could be used for input and output. This would provide named files for the tool to open, but we had to abandon this idea. First, lack of local storage on some systems forced us to resort to the distributed file system, but we had little control over when data is flushed to disk. Ideally, no data would be flushed to disk, we just wanted a file-like view of the already resident data. The pipes were plagued by buffering issues. None of the known facilities to remove the buffering (`setvbuf`, `fadvise`, etc.) worked reliably. We needed something more predictable; it was time to move on.

A solution that works

At this point, we began to look at shared memory segments. We started with System V shared memory segments as seen in the classic System V IPC library. HSP-Wrap on the worker node would create anonymous SHMs for all of the shared data files. `libstdiowrap` would correlate file paths to SHMs via some lookup table (stored in another SHM) or by querying the HSP-Wrap process via another IPC channel. Preliminary testing showed that this technique is feasible, but we still need a method to to stream input and output to the process. We attempted to use pipes, as available through the `pipe` system call, but ran into the same problems we faced with the named pipes approach above. Kraken exhibited particularly strange behavior with the pipes resulting in corruption.

The decision to support streaming wasn't actually realized at this point in development. Thus, our first version of the wrapper, coined MCW for the "Master, Controller, Worker" hierarchy, operated with System V SHMs as the only IPC mechanism [3]. The controller would allocate a SHM for each shared file as well as a SHM for each process's input and output file. The controller would copy the input to the process's input SHM, execute the tool, and copy the results from the output SHM upon completion. The controller would then replace the input with the next and re-execute the process. We explain why this is not ideal in the next section. This solution allowed us to scale PSI-BLAST on Kraken for the first time. A comparison of the MCW technique and the baseline approach are presented in figure 3.4 as *HSP-BLAST (non-streaming)* and *NCBI-BLAST*, respectively. The performance is much improved as it approaches near linear scalability, but also relies on additional optimizations described later.

We set out to test MCW on the Beacon system and quickly realized that; unlike Kraken,

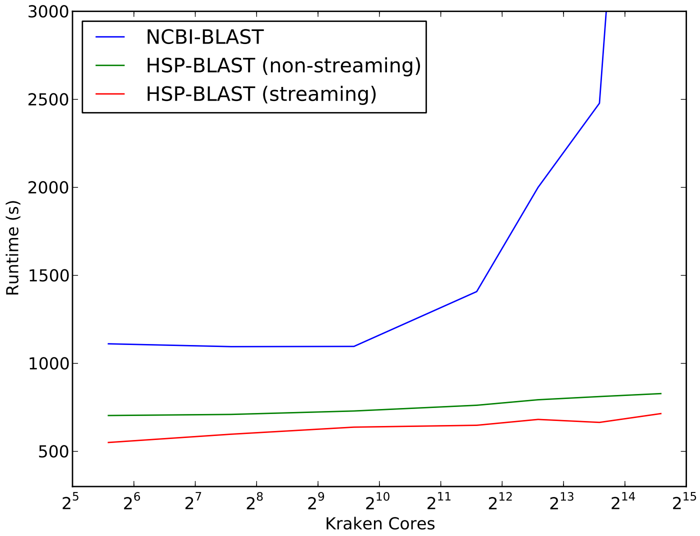


Figure 3.4: Weak scaling comparison of the original NCBI, non-streaming HSP, and streaming HSP BLAST runtimes

Newton, and other systems we tested; System V SHMs are not supported. POSIX Shared Memory Segments seemed to be appropriate, and we found them to be adequate after testing some skeleton code on our target platforms. We were also faced with new requirements at this point: support for multiple input and output files, fault tolerance, Intel MIC support, among others. This is when we decided to rewrite MCW as HSP-Wrap to support POSIX SHMs, prepare for new requirements, and to finally solve the streaming problem.

A better solution: streaming

We persisted with the POSIX IPC mechanisms since they seemed to be well supported on all of our targets. We could have possibly used UNIX local domain sockets to stream data between the threads, but why bother adding an additional environmental requirement to the software? At this stage, after all our frustration in finding a widely-supported IPC technique, we wanted to continue with what we know works. We still needed some way to synchronize the processes and that is where the POSIX mutexes and condition variables fill the gap.

We centralized the synchronization primitives, process control information, and file table into a single structure named the process control block. The block is stored in a SHM shared between HSP-Wrap and all of the node's worker processes. The layout is shown in figure 3.5.

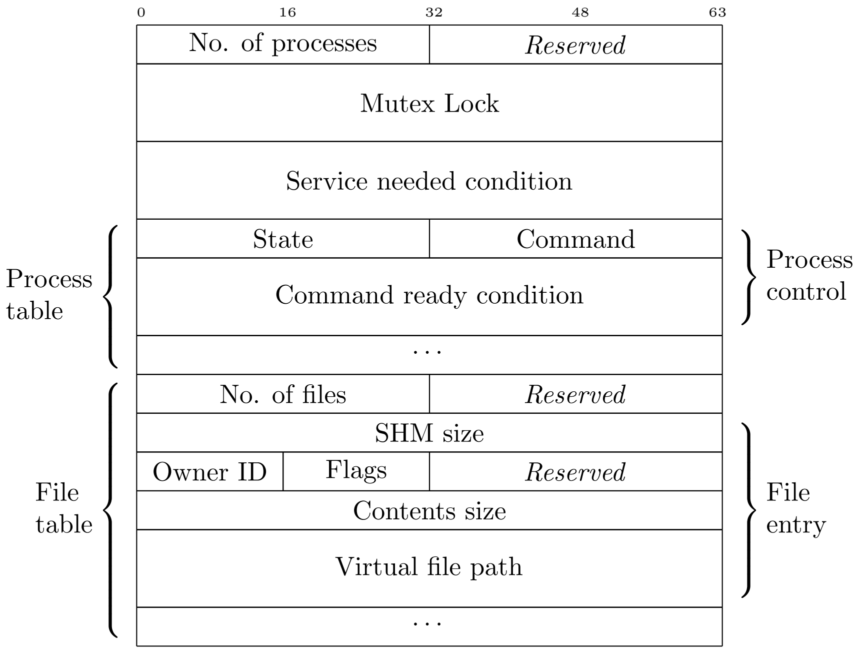


Figure 3.5: The layout of the process control block

This structure is essentially the application binary interface (ABI) between HSP-Wrap and `libstdiowrap`, and thus, any software conforming to this ABI could substitute as the wrapper harness or as the distributed tool.

Regarding the structure, there is a lock used to provide mutually exclusive access to portions of the block. The "service needed" condition variable is used by the worker processes to signal HSP-Wrap when their status has changed. This allows HSP-Wrap to wait until a worker needs service. The process table contains a control structure for each worker process. This includes the state of the process as reported by `libstdiowrap`, and a command issued by HSP-Wrap. A process in need of service is able to wait on its "command ready" condition variable until HSP-Wrap has serviced its request.

The file table indexes all of the virtual files. `libstdiowrap` opens a file by looking up the file path in the table. If the shared flag is set, then the SHM corresponding to the index of the entry is mapped. If the file is not shared, then the file entry's owner ID must match the ID of the worker process. This is because each of the unshared files has duplicate entries in the table; one for each worker process. The worker process is responsible for locating the SHM dedicated to the $\langle worker, filepath \rangle$ pair and mapping it. Our solution supports multiple input and output streams per process partially due to this multidimensional association.

MCW only supported one input and one output file per process.

The worker process reads from the input buffers until there is no more data. The amount of data read accounts for one, but possibly more, input records. At this point, the worker must set its status to `PS_EOD` to indicate end-of-data, signal the service needed condition variable, and wait on command ready. HSP-Wrap can then copy new input into the buffer, set the command to `PC_RUN` and signal command ready. `libstdiowrap` realizes if no data was copied into the buffer and reports the file as `EOF`. Similarly, for output, the worker sets the status to `PS_NOSPACE` when the buffer is full. HSP-Wrap queues the buffered output for writing, and sets the command to `PC_RUN` to resume the process with an empty buffer. HSP-Wrap can also issue the `PC_QUIT` command if the process should terminate.

The addition of streaming brought a pleasant performance improvement. We found that the streaming solution of NCBI BLAST is 10% to 20% faster for typical work loads. A reflection of this is seen in figure 3.4 as *HSP-BLAST (streaming)*. Of course the improvement varies depending on how much time is taken to initialize the tool versus the time taken to process a single input, but any speedup is desirable.

3.5.2 Process pool

The non-streaming HSP solution described earlier worked well, but it faced unexpected failures at times. Nodes would occasionally fail with a bus error (`SIGBUS`) on Kraken, and the issue was even more prevalent when we tested on Newton. We comprehensively reviewed the source code, and even wrote a simplified test case which also failed unpredictably. We began to review documentation for the MPICH2 and Open MPI implementations of the MPI specification. It turns out that, the MPI interface is thread safe, but implementations are not required to be thread safe. Many implementations seem to have particular issues with spawning a new process with `fork` and `exec` after the MPI library has been initialized [42,43]. For example, we received this error message when running with Open MPI on the Newton cluster:

```
An MPI process has executed an operation involving a call to the
"fork()" system call to create a child process.  Open MPI is currently
operating in a condition that could result in memory corruption or
other system errors; your MPI job may hang, crash, or produce silent
data corruption.  The use of fork() (or system() or other calls that
create child processes) is strongly discouraged.
```

The root cause is when the MPI process calls some unsafe function (including MPI functions, `malloc`, `fork`, and a host of others) between calls to `fork` and `exec`. MCW was particularly vulnerable because it forks new processes frequently, one for each input. A simple work around is shown below:

Listing 3.2: Naive workaround for fork/exec race condition

```
/* Start child */
pid t pid = fork();

if (pid == 0) {
  /* Child */
  if (execve(exe, argv, env_list)) {
    fprintf(stderr , "Could not exec: %s\n", strerror(errno));
    exit(EXIT_FAILURE);
  }
} else if (pid > 0) {
  /* Parent */
  /* We must wait a second for the child to complete the execve,
     otherwise, this parent process might call an unsafe function
     between the fork() and exec() invocation. */
  sleep (1);

  /* Now we can get on with our life... */
}
```

The trick is to simply delay the parent process (the node's MCW controller process) until the `exec` call has most likely completed. This is not a correct solution since there is no guarantee that `exec` has completed in the time allotted, and we risk wasting too much time waiting if the timeout is set to a less optimistic value. This trick was enhanced by instead forcing the parent to wait for a user signal (`SIGUSR1`) instead of sleeping. `libstdiowrap` sends the signal to the parent process once it has initialized; confirming that the new process has, in fact, started. This worked on Kraken and other computers, but receipt of the signal would cause the process to crash on Beacon.

HSP-Wrap is less likely to experience this issue since it only spawns one process for each CPU core, versus one for each incoming input record. That fact does not make executing the tools on HSP-Wrap foolproof. We would also like to reserve the capacity to fork a different tool process during the execution; for example, to restart failed processes or to migrate a CPU core to a new tool. If the source of the problem is calling `exec` from one of the MPI process's children, then we can completely avoid this situation by attempting to `fork` and `exec` from a process unassociated with the MPI process.

This discovery led to the concept of a daemonized process pool. The process pool is preforked and reparented to the UNIX `init` process before the main process runs any MPI functions. The main MPI-enabled HSP-Wrap process is then able to fork new processes by sending commands to the process pool. Again, we chose to use POSIX IPC stored in another shared control block (the process pool control block). The mechanism allows for the process pool to report when it is ready (so that the parent process can wait until the pool is properly daemonized), and to accept requests to spawn another process. The resulting schematic is

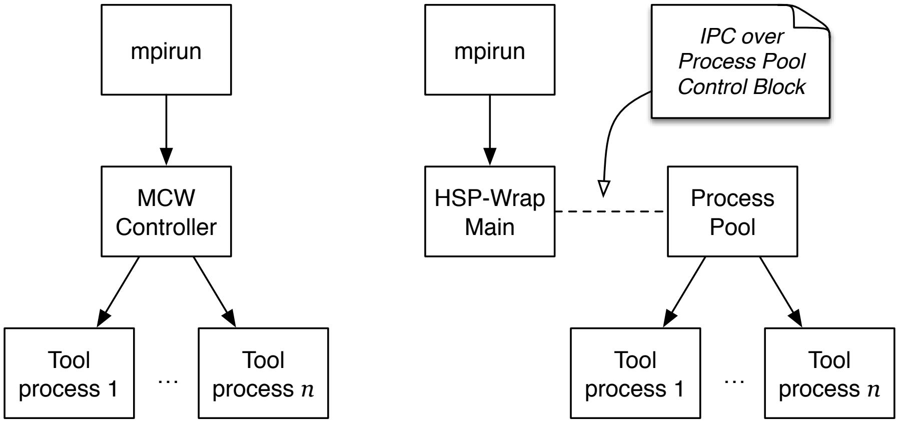


Figure 3.6: Comparison of MCW and HSP-Wrap process trees for a single worker node

shown and compared to the MCW approach in figure 3.6. The method to daemonize the process consists of forking an intermediate process, then forking the process pool from the intermediate process. We then kill the intermediate process; causing the process pool to reparent to the `init` process. The process is then placed in a new session through a call to `setsid`.

3.5.3 File broadcasting

Our preliminary profiling showed an exponential slowdown as the number of worker nodes increases (figure 3.1). Further investigation suggests that this is largely due to contention for the large shared database file; more workers implies more contention. Therefore, we wanted to optimize the behavior of reading shared files from storage to prevent these redundant loads. This is a responsibility of the master node, as implied earlier. The master reads the shared files from disk then broadcasts them to the worker nodes through a single collective MPI call (`MPI_Bcast`). This distribution is very efficient and scales logarithmically to the number of nodes.

Chunked transfers

MCW would read the file from storage by memory mapping it, then broadcasting the entire segment. Some MPI implementations have issues with accessing memory mapped regions. In order to remedy this, we must use some intermediate buffer. We studied the performance of a number of reading strategies across a range of different buffer sizes. The buffer size determines how much file data is stored intermediately before sending the data via `MPI_Bcast`.

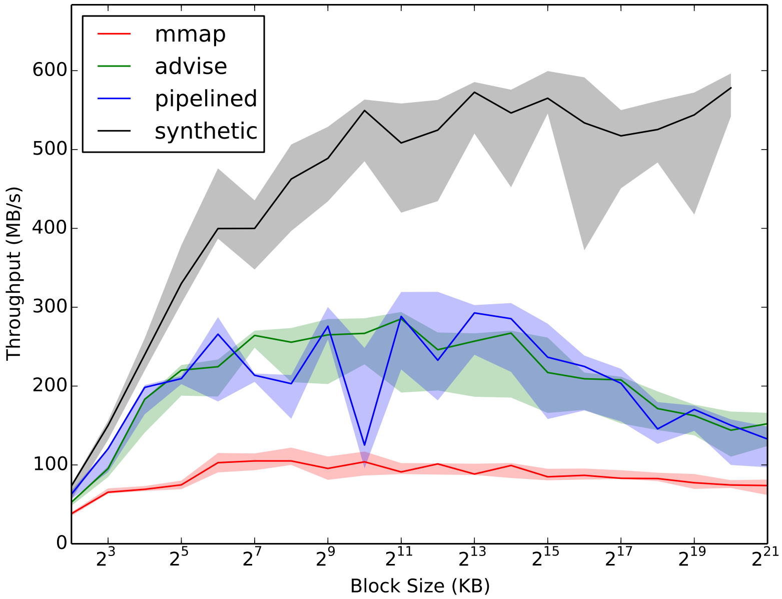


Figure 3.7: A comparison of throughput for different broadcast strategies and transfer block sizes

We denote this size as the "chunk size". The reading strategies include using `mmap` to access the file data, using the `read` system call combined with hints to the `fadvise` call, and to use an extra thread to pipeline the reading and broadcasting. We also measured a synthetic maximum where the data is not read from disk at all, but the chunks are instead generated in memory before broadcasting. The results of this study, as performed on the Beacon machine, are displayed in figure 3.7.

What we see is the memory mapping ("mmap") approach is the slowest, and this is not necessarily a surprise. The operating system must page the data into memory, and even with mechanisms such as `madvise`, we do not gain the same control as explicitly reading when we want the data. The "advise" approach uses `fadvise` to mark the entire file as `POSIX_FADV_SEQUENTIAL` to let the operating system know we will read the file sequentially. We also mark the next chunk as `POSIX_FADV_WILLNEED` before broadcasting the current chunk; allowing the OS to optimistically fetch the chunk before we call `read`. The chunk is marked as `POSIX_FADV_DONTNEED` to aggressively purge any cached data we no longer need. The threaded "pipelined" approach performed similarly to "advise", which is better than our baseline "mmap", but with greater variance.

All approaches have terrible performance with very small chunk sizes. This is due to the overhead of sending each broadcast, which overshadows the time actually spent transferring the payload. Very large block sizes (in the 1 GB range, which approaches the size of an entire file) perform worse than moderately sized blocks. This can be explained due to the master needing to read the entire file before sending the whole file; there is no opportunity to concurrently read data from storage while broadcasting. We opted to use the "advise" solution since it performs on par with the "pipelined" solution, but with less complexity. The default chunk size used by HSP-Wrap is 1 MB, although this can be changed to tune the software for different systems. What started as a trial to work around an MPI limitation turned out to be an opportunity to improve the performance further.

3.5.4 File distribution

The actual data parallelism of HSP-Wrap is implemented through the file distribution strategy. The inputs need to be distributed across the worker nodes so that each input is processed once, and so that the processor cores of the worker nodes stay as saturated as possible. The strategy must dynamically load balance the input as determined in section 3.2.1. We reviewed a number of approaches such as work stealing, but we decided to elect a hierarchical approach with the master situated at the top. This is because a single master helps to offload managerial tasks off of the worker nodes and this scheme is a good-fit for the file broadcasting system we already had in place. The master must be able to keep pace with the demand from the subordinates in need, but we have not seen this to be an issue in practice.

Centralized dispatch

The process begins with the master, which after broadcasting the shared files, creates an iterator for each input file. An iterator is a parser designed to walk through units of work for a particular file format. HSP-Wrap ships with a number of iterators designed for sequences of a FASTA file, compounds of a MOL2 file, and lines of a plain-text file. More iterators can can be added if support for other file types is required. The master then enters the servicing mode, where it services requests for more work from the worker nodes. The master uses the iterators to select an extent of work units, which are then sent to the worker node in the master's response. It is this request-oriented system that helps to ensure an even load balance.

Prefetching

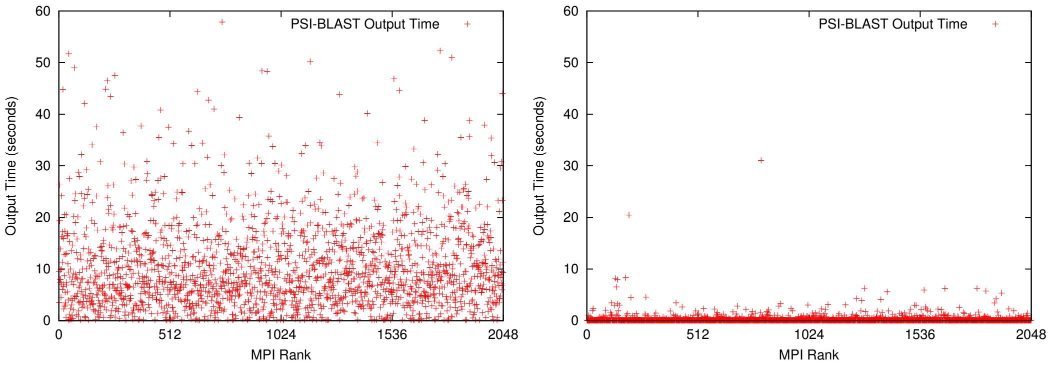
Moving down the hierarchy one level, we have HSP-Wrap running on the worker nodes. The goal of the worker node, besides executing the tool processes, is to serve the same function as a cache found in a traditional memory hierarchy. Instead of reducing the time to access data from main memory, however, we reduce the time to access input data from the master node. HSP-Wrap tries to keep the node's local work queue full by prefetching work from the master. The intent is to hide all input latency introduced by communication with the master node. The desired fullness of the queue is tunable. We recommend queuing at least as many units of work as there are CPU cores in order to handle the worst case scenario where all tool processes need work at approximately the same time. Tools that process their input very quickly, such as MUSCLE, can benefit from an even deeper queue.

3.5.5 File output

The second half of the I/O problem is naturally the output. HSP-Wrap aims to improve output routines by providing enhanced performance and improved reliability. We stick with the MPI recommended theme of overlapping computation with communication. To this end, HSP-Wrap utilizes additional threads dedicated to writing results to storage. These threads are referred to as the writers. Each node maintains a writer for each of the output files. The writers are shared by the tool processes hosted on that node. Each writer employs a two-stage buffering technique. When a tool process generates output, the output is stored into the SHM buffer designated to the process for that output file stream. When this buffer is flushed, due to it becoming full or the process exiting, the contents are immediately copied to the writer's front buffer for eventual write via a simple `memcpy` operation. This effectively implements an asynchronous write, hiding output latency.

The writer actually writes the results to disk once the front buffer approaches a full state or if a substantial amount of time has elapsed since the previous write. This is implemented by a buffer swap, where the front buffer used to store incoming data is swapped with the back buffer. We chose this method in order to reduce the time spent in the critical section. The tool processes and the writer thread potentially block only for the time needed to swap two pointer values. The writer thread is then signaled in order to finally write the back buffer to storage.

The solution goes further than simply overlapping computation and communication, however. The use of a single writer thread for the compute cores of a worker node help to reduce the contention for the storage resource since the requests for a node are now bundled. Similarly, the amount of overhead is reduced by utilizing larger writes instead of many smaller writes. The Lustre file system, at least, is able to perform a parallel write across multiple object storage targets (OSTs) when a sufficient amount of data is provided, boosting output bandwidth. Finally, HSP-Wrap arranges the output files in a hierarchical directory layout which allows parallel use of the OSTs [44].



(a) Without background writer process.

(b) With background writer process.

Figure 3.8: A comparison of time spent blocking on output before and after implementing a background writer process

Figure 3.8 shows the total time spent writing data to storage by each of the writer threads. The highly variant output times for the original non-optimized solution are presented in figure 3.8a. Figure 3.8b shows the dramatic increase in performance afforded through the output optimizations. Not only is the total output time greatly reduced, but the times are more uniform and predictable as well.

3.5.6 Fault-tolerance

HSP-Wrap implements a number of features to make the wrapped tool more resilient in the event of failure. The process pool, with its ability to spawn a new process during the execution of a job, provides the means to restart a tool process if the tool fails. This could be due to erroneous input, a bug in the software, or a transient system failure of some sort. This functionality allows for the job to continue utilizing all of the compute cores instead of degrading or failing altogether.

HSP-Wrap's writer threads are critical to our fault-tolerance implementation. For one, the writer's behavior of flushing data after a timeout helps to enforce an upper limit on data loss if there is a system failure, the job's maximum wall time is reached, or a tool process hangs and becomes unresponsive. The primary fault-tolerance feature, though, is the support for journaling.

There is an additional file, the index, associated with each writer to complement the primary output data file. A journal entry is written to the index any time the writer swaps the output buffers to prepare for an asynchronous write. The format of a single entry is illustrated in figure 3.9. The index file maintains a map between the physical file and the

Stream ID	Virtual stream offset	Physical file offset	Length	CRC

Figure 3.9: The layout of a journal entry for a single output block

virtual streams. Such a mechanism is necessary since the output file contains the multiplexed output of each of the node's tool processes. The *Stream ID* is simply the ID of the tool process which generated this block of resultant output. The position of this block within the logical output of the specified *Stream ID* is stored in the *Virtual stream offset*. The *Physical file offset*, on the other hand, holds the position of the block as stored within the shared output file. The *Length* is the size of the block, in bytes. The length is the same for both the virtual view and the physical file on the file system. Finally, the *CRC* field stores a 32 bit CRC checksum of the data block.

The index file is used by the operator to gather output after the job completes. The output file is traversed to retrieve all of the output, or the output of a specific stream. The CRC value is checked to detect errors in the output data or in the index itself. Regardless of whether log or output is corrupt, we can reject the result due to the discrepancy.

The MPI runtime environments on the systems we tested do not survive hardware failure gracefully. That is, the entire job will abort if a node goes off-line or a network link fails. There were no provisions, at the time of experimentation, to recover from such an error. In response to this, HSP-Wrap strives to minimize data loss by providing a resume facility. The operator must run another job identical to the one that failed, but must specify the resume option. HSP-Wrap replays the journal and verifies the data stored in the output files. From this, HSP-Wrap can compute the difference between what was intended to execute and what outputs are actually properly stored. After the replay, HSP-Wrap can run as it typically does, but only distributing work units that are missing from the original result set.

3.5.7 Heterogeneous system support

This section describes how HSP-Wrap manages to support heterogeneous systems. In particular, we focus on the symmetric mode provided by the Intel MIC (Xeon Phi) co-processors hosted on the Beacon supercomputer. The symmetric mode operates by exposing the Xeon Phi cards as additional MPI ranks. HSP-Wrap allocates each card as an additional worker node, and continues to operate with the same principles described throughout this chapter. Xeon and Xeon Phi processors do not have binary compatibility, this means that two HSP-Wrap programs must be compiled. In addition, we must compile two binaries of the tool (such as BLAST or DOCK6). HSP-Wrap sends both binaries from the master node to all of the worker nodes during the shared file broadcasting stage. The worker nodes determine the architecture of their node, select the appropriate executable, and discard the other one. All nodes now have the correct binary and can start running input delivered from the master.

HSP-Wrap can also run in a hybrid mode. In this case, HSP-Wrap operates exactly as it does in the homogeneous systems described in earlier sections. However, the tool binaries themselves are enhanced to take advantage of the Xeon Phi co-processors. This allows for course-grained scaling of the problem through load balancing and fine-grained scaling through vectorization and offloading of computation from the CPU to the co-processors.

3.5.8 Using HSP-Wrap

The system administrator or operator must first build and optionally install the software in order to make it available on a system. Both the HSP-Wrap binary and the wrapped informatics software need to be compiled for all target architectures in use by the system, for example `x86_64` and `mic`. `libstdiowrap` and other support libraries are statically linked with the binaries. This installation procedure only needs to be performed once or when the software needs to be updated. The administrator can make the files available to all users on the system, which is what we did on Kraken with the cooperation of the NICS User Support team. A user can also simply copy the binaries to their user-owned work space.

Preparing the job

A user prepares a job by first choosing or creating a directory from which to run the job. The run directory must be located in distributed storage for all the systems we targeted. The user must also copy or generate their input and shared files and place them in a location accessible by the compute nodes (usually the user's Lustre "scratch space"). These input files can be shared by multiple jobs if desired. For example, we maintained copies of the popular nr, nt, and pfam databases on Kraken for use by all HSP users. The user then creates a PBS (job script) file in the run directory to describe the job. The PBS file is not a requirement of HSP-Wrap, but it is the method of submitting non-interactive jobs on all of the HPC resources we used. An example PBS file follows.

Listing 3.3: An example PBS script with embedded HSP-Wrap configuration

```
#!/bin/bash
#PBS -N example-job
#PBS -l walltime=0:30:00,size=96

# Choose the right hspwrap binary based on architecture
```

```
case $(uname -p) in
  *x86_64*)
    hspwrap_bin="hspwrap-xeon" ;;
  *slesmic*)
    hspwrap_bin="hspwrap-mic" ;;
  *)
    exit 1 ;;
esac

# Determine number of nodes on Kraken
cores=12
nodes=$((PBS_NNODES / cores))

# Chdir to run directory
cd $PBS_O_WORKDIR

# Use aprun to start hspwrap with configuration
aprun -n$nodes -N1 -d$cores $hspwrap-bin <<HSP-CONFIG
  # Optional
  master {
    bcast-chunk-size 4M;
  };

  phase one {
    arch x86_64 {
      exe-file "blastall-xeon";
    };
    arch mic {
      exe-file "blastall-mic";
    };
    # The arguments can also be specified per-architecture
    exe-args "-p blastp -i input.fasta -o results.blast";

    stream "input.fasta" {
      path "inputs-1440.fasta";
      format fasta;
      # input is default
    };

    stream "results.blast" {
      output;
      # path is inferred by stream name
    };

    # map all files in nr-5m with 1:1 naming
    file "/lustre/scratch/proj/sw/hsp/data/nr-5m/*";
  };
HSP-CONFIG
```

The PBS file contains shell code scaffolding that varies across the different HPC machines. The script changes the working directory to the run directory, determines which HSP-Wrap binary should be used for the machine, then runs `hspwrap` through `aprun` (`mpiexec` on most systems) with the specified HSP-Wrap configuration. The configuration is fed to HSP-Wrap through standard input as a means to consolidate the configuration and to prevent opening an additional configuration file. Alternatively, a configuration filename can be specified as the first argument to `hspwrap`. Earlier versions of the wrapper used to accept configuration through many environment variables. This technique resulted in long colon-delimited strings and became unwieldy once multiple input files, output files, and various tuning options were added. Instead, we use an extensible grammar implemented in Flex and Bison which provides a more readable format with better error checking. We try to adopt a "convention over configuration" methodology to keep the most common use cases simple while still providing configurability for more specialized cases. The grammar in Backus-Naur Form (BNF) is listed below.

Listing 3.4: The configuration grammar used by HSP-Wrap

```
⟨commands⟩           ::= λ | ⟨commands⟩ ⟨command⟩ ‘;’

⟨command⟩            ::= ⟨master-set⟩
                      |  ⟨phase-set⟩

⟨master-set⟩         ::= ‘master’ ‘{’ ⟨master-statements⟩ ‘}’

⟨master-statements⟩  ::= λ | ⟨master-statements⟩ ⟨master-statement⟩ ‘;’

⟨master-statement⟩   ::= ‘bcast-chunk-size’ ⟨size⟩

⟨arch-set⟩           ::= ‘arch’ ⟨word⟩ ‘{’ ⟨arch-statements⟩ ‘}’

⟨arch-statements⟩    ::= λ | ⟨arch-statements⟩ ⟨arch-statement⟩ ‘;’

⟨arch-statement⟩     ::= ‘exe-file’ ⟨string⟩
                      |  ‘exe-args’ ⟨string⟩

⟨phase-set⟩          ::= ‘phase’ ⟨word⟩ ‘{’ ⟨phase-statements⟩ ‘}’

⟨phase-statements⟩   ::= λ | ⟨phase-statements⟩ ⟨phase-statement⟩ ‘;’

⟨phase-statement⟩    ::= ⟨arch-statement⟩
                      |  ‘depends-on’ ‘phase’ ⟨word⟩
                      |  ⟨arch-set⟩
                      |  ⟨stream-set⟩
                      |  ⟨file-set⟩

⟨stream-set⟩         ::= ‘stream’ ⟨string⟩ ‘{’ ⟨stream-statements⟩ ‘}’
                      |  ‘stream’ ⟨string⟩

⟨stream-statements⟩  ::= λ | ⟨stream-statements⟩ ⟨stream-statement⟩ ‘;’
```

```
⟨stream-statement⟩     ::= ‘path’ ⟨string⟩
                         | ‘input’
                         | ‘output’
                         | ‘format’ ⟨word⟩
                         | ‘prefetch-ratio’ ⟨number⟩

⟨file-set⟩             ::= ‘file’ ⟨string⟩ ‘{’ ⟨file-statements⟩ ‘}’
                         | ‘file’ ⟨string⟩

⟨file-statements⟩      ::= λ | ⟨file-statements⟩ ⟨file-statement⟩ ‘;’

⟨file-statement⟩       ::= ‘path’ ⟨string⟩

⟨number⟩               ::= ⟨digits⟩ ‘.’ ⟨digits⟩ | ⟨digits⟩

⟨size⟩                 ::= ⟨digits⟩ ⟨suffix⟩ | ⟨digits⟩

⟨digits⟩               ::= ⟨digit⟩ ⟨digits⟩ | ⟨digit⟩

⟨suffix⟩               ::= ‘K’ | ‘k’ | ‘M’ | ‘m’ | ‘G’ | ‘g’

⟨word⟩                 ::= ⟨alpha⟩ | ⟨alpha⟩ ⟨word-chars⟩

⟨word-chars⟩           ::= ⟨word-char⟩ ⟨word-chars⟩ | ⟨word-char⟩

⟨word-char⟩            ::= ⟨alpha⟩ | ⟨digit⟩ | ‘_’ | ‘-’ | ‘.’

⟨string⟩               ::= ‘"’ ⟨string-chars⟩ ‘"’

⟨string-chars⟩         ::= λ | ⟨string-chars⟩ ⟨string-char⟩

⟨string-char⟩          ::= ‘\"’ | ‘\\’ | all characters except ‘"’ and ‘\’
```

Running the job

The user runs the job just as they would queue any other batch script. On most systems, the job is queued with the `qsub` command:

```
qsub -A account example-job.pbs
```

Upon completion of the job, the user will see a subdirectory in the run directory containing the results and some temporary files used for scratch. These results are not immediately usable because the outputs are spread across multiple files in a hierarchical directory structure, one file per output stream per node. Each file contain interleaved data from the worker processes which is interpreted by referencing the companion index file. We provide a tool aptly named `hspgather` to aide in reading the program output. The help message for HSP Gather and an example that pipes results to a user script for processing is listed below:

```
$ hspgather --help
Usage: hspgather [OPTION]... FILENAME

HSP Gather is the standard way to coalesce the output results of a
completed HSP job.  The resulting output files with the name FILENAME are
deinterleaved and concatenated to standard output, unless the -o option
is specified.

Options:
 -d, --directory=DIR the directory to use when searching for result files
 -f, --force         forcibly overwrite output file if it already exists
 -i, --ignore-errors print a warning instead of failing if errors occur
 -o, --output=FILE   write to output file instead of standard output
     --help          display this help and exit
     --version       display version information and exit

$ hspgather results.blast | ./user-script.sh
```

The suite of tools supplied with the distribution (`hspwrap`, `gather`, and `recover`) provide the user with the operations they need to run distributed tools, process the resultant data, and to recover output from a failed job. Tools can be added or expanded as user requirement surface, but so far these tools have satisfied our needs.

Chapter 4

Results

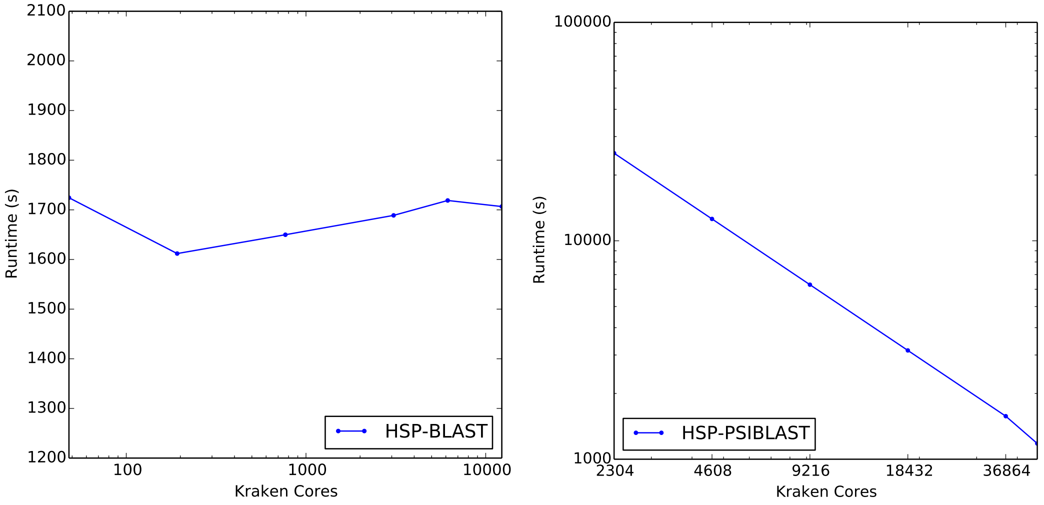
The timing results of the preliminary scalability study in figure 3.4 show encouraging performance for an artificial work load. In that study, we duplicated the input sequences in proportion to the increase in the number of cores. Our motivation for choosing inputs in this fashion is to provide a fair comparison of the statically distributed original NCBI code and the wrapped HSP solutions. If each processor was statically assigned a different set of sequences then some processing cores may complete their workload much earlier than their peers, wasting cycles due to idling at the end of the job. These results, although promising, may not speak to the performance of HSP-Wrap when coupled with real world input. We worked with researchers in the fields of bioinformatics and cheminformatics, and used HSP-Wrap to process their data on HPC resources. This chapter highlights the performance results of a number of these collaborations as well as the performance across several HPC installations.

4.1 Reusability

In this section we present scalability testing results for individual informatics software programs. The goal of this section is to show how the HSP-Wrap software is a useful utility for scaling different programs to HPC resources. The machine used through most of this section is Kraken due to the time allocation available to us on this resource.

4.1.1 NCBI BLAST

Optimization of BLAST is the original focus of our research. Scalability studies for BLAST were repeatedly tested on many machines throughout the development of the software in



(a) Weak scaling study of BLAST

(b) Strong scaling study of PSI-BLAST with 1 million sequence searches

Figure 4.1: Graphs showing the scalability of the wrapped BLAST and PSI-BLAST programs

order to assess the performance impact of various features as they were introduced. The computational results of these tests were compared with the results of the unmodified BLAST code in order to verify correctness.

Weak scaling results

We present the performance results of one of the most recent weak scaling studies in figure 4.1a. The Protein BLAST function (`blastall -p blastp`) was used to run a search for 194 560 input protein sequences against the non-redundant (*nr*) protein database on Kraken with 12 288 cores. The input sequences are randomly sampled from a real request submitted by one of our collaborators. The job was then repeated on fewer cores with a proportionally smaller number of input sequences; this setup is listed in table 4.1 below.

The results show acceptable scaling results. The characteristics of the curve closely match the shape of the HSP-BLAST curves from the artificial study expressed in figure 3.4. The runtime is fairly constant across all test points. The inflection seen in the unoptimized naive experiment is absent from both the preliminary wrapped results as well as this real world study. This suggests that the improvement is due to our optimizations provided in the wrapper software, and the improvements are not due to some other mechanism such as local caching of duplicated input queries as may have been the case in the preliminary study.

Table 4.1: Experiment setup and resulting total runtime for the referenced BLAST weak scaling study.

Cores	(Nodes)	Sequences	**Runtime (s)**
48	4	760	1724.39
192	16	3040	1611.94
768	64	12 160	1649.85
3072	256	48 640	1688.84
6144	512	97 280	1718.87
12 288	1024	194 560	1706.65

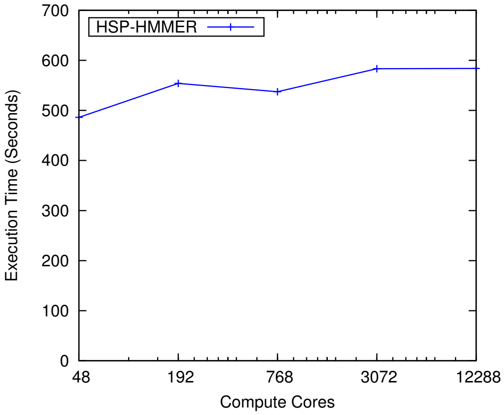


Figure 4.2: Weak scaling study of HMMER

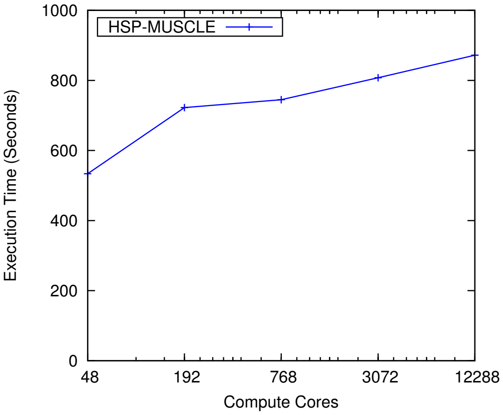


Figure 4.3: Weak scaling study of MUSCLE

Strong scaling results

One may argue that weak scaling is not the most meaningful way to measure scalability. In particular: a weak scaling study either requires the problem to be duplicated for each compute resource (which doesn't accurately model a real usage scenario), or the problem needs to be sampled to maintain the ratio. This sampling may introduce error since the smaller samples may no longer accurately represent the original corpus. Therefore, we also present strong scaling results.

A graph showing the scalability of PSI-BLAST using a strong scaling methodology is displayed in figure 4.1b. Each test point represents the runtime for our wrapped PSI-BLAST software to complete 1 million search queries on Kraken with the specified number of compute nodes. Results from this study shows near linear scalability.

4.1.2 HMMER

We assessed HMMER's `hmmscan` tool in performing searches against the 24.0 version of the Pfam database. Sequences with 250 – 450 Amino Acids were selected from the *nr* database. Our selection for the number of sequences per core is higher than what we used for BLAST due to the comparatively fast query speed of HMMER. The weak scaling results shown in figure 4.2 represent the runtime of HMMER on Kraken where the input file contains 1000 for each core (as opposed to 120 sequences per core used in the BLAST weak scaling study). Again, we see a near linear speedup for these large input query counts.

4.1.3 MUSCLE

The performance of the wrapped version of the MUSCLE multiple sequence alignment tool (`muscle`) was evaluated on the Kraken supercomputer. We chose 10 input data sets for each core used in the experiment. Each data set consists of sequences returned by previous BLAST alignment searches. Figure 4.3 shows the scaling results of MUSCLE up to 12 288 cores on Kraken. MUSCLE does not load any search database, unlike BLAST and HMMER, but instead aligns the sequences contained in each of the data sets. These results, therefore, reflect the performance of a tool where the shared file distribution is not used. However, the tool does leverage HSP-Wrap's dynamic load balancing and optimized file output features. The performance over a naive solution is unknown to us, but the performance is acceptable. The ease at which HSP-Wrap is able to distribute the workload and prepare results makes HSP-MUSCLE a useful tool nonetheless.

4.1.4 DOCK6

DOCK6, our cheminformatics use case, also faired well when used with the HSP-Wrap software. The data used for our DOCK6 scaling study are taken from a real world collaboration with the Medical University of South Carolina. The original study involved docking 1 400 000 compounds from the ZINC database onto 4 proteins: arrestin, RGS4, a G-protein (Ras), and a peptidoglycan synthase from bacteria (PBP2). The purpose of the study was to test the accuracy of DOCK6 when used for drug discovery as well as whether the performance of docking at this scale is viable on HPC resources.

For our scaling study, we sampled inputs from the ZINC compounds used for the original study. These input compounds are then docked onto the Ras protein, and the results are written to storage. We tested the scalability from 120 cores to 3840 cores on Kraken. The number of compounds used for docking range from 5000 to 160 000, that is $41\frac{2}{3}$ compounds per core. Figure 4.4 shows the scaling results. Again, the speedup is not linear, but it has a

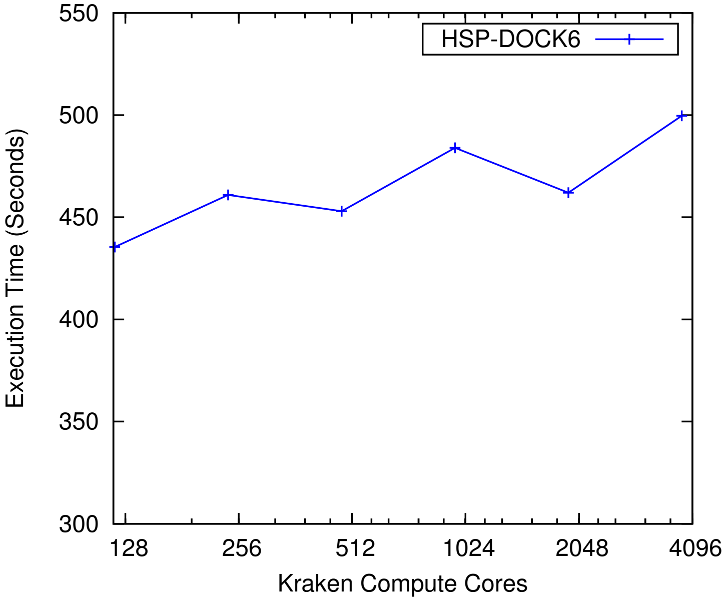


Figure 4.4: Weak scaling study of DOCK6

linear trend. This is much better than the super-linear performance seen in the unwrapped BLAST evaluation.

4.2 Effectiveness with Intel MIC

We briefly evaluated the performance of BLAST using the Xeon Phi co-processors on the Beacon cluster. Two test cases were evaluated for scalability: computing only on the Xeon Phi MIC cores and running on a combination of MIC and the host Xeon cores. The jobs are configured such that each Xeon Phi card acts as an additional MPI rank (that is, an HSP worker node per card). The HSP-Wrap process on each card manages a BLAST process for each of the card's MIC cores. We realize that this is not an ideal way to leverage the Xeon Phi co-processors as we do not take advantage of the vectorization units, but instead rely on coarse-grained parallelism. The intent of the study is to instead determine whether HSP-Wrap works on a processing architecture other than x86-64, and to demonstrate how HSP-Wrap can distribute a job across heterogeneous compute nodes.

For this weak scaling experiment we selected 300 sequences per core for each data point. The MIC tests are performed on 1, 2, 3, and 4 Xeon Phi cards housed on a single Beacon node, then 8, 16, and 32 cards by increasing the number of Beacon nodes to 2, 4, and 8

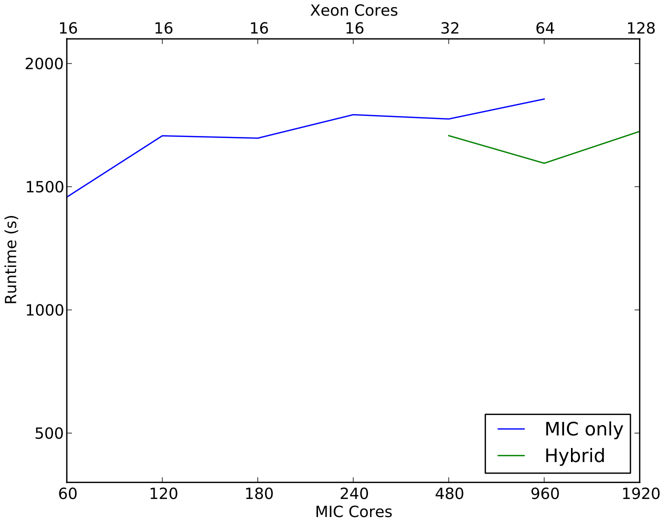


Figure 4.5: BLAST on Beacon

respectively. There are 60 MIC cores per Xeon Phi card. We also test the hybrid performance by repeating the tests with the additional Beacon nodes serving as Intel Xeon worker nodes instead of simply as a host for the Xeon Phi cards. Each Beacon node has 16 Xeon Cores, however there is no data for 1 node jobs since the node is dedicated as master in these cases. All binaries are compiled with Intel's `icc` and `mpiicc` compiler wrapper.

Figure 4.5 shows the resulting performance. These results should not be interpreted as a comparison of the Xeon and MIC processors, but should instead be viewed as a representation of HSP-Wrap's applicability on a different and mixed computer architecture. We see near linear scaling for both the MIC-only and the hybrid configurations. These figures suggest that HSP-Wrap is well-suited for the Beacon environment. In particular, this system differs from Kraken in the use of a different Lustre installation (Medusa), MPI implementation, processor (Intel Xeon), and interconnect (Infiniband). The MIC results show near linear scalability, suggesting that HSP-Wrap can be applied to different platforms. However, a solution using HSP-Wrap to distribute the work to the Xeon cores, then using a technology such as OpenACC to parallelize regions of code onto the co-processors in Offload mode may yield better results.

Chapter 5

Conclusion

This paper describes one way that that high performance computing resources can be used for scientific discovery. We demonstrate how the use of efficient I/O management combined with dynamic load balancing can scale a number of software tools to tens of thousands of nodes on several HPC installations. Through the use of locally cached data, compression, on-demand scheduling, and buffering we are able to significantly improve the performance of these jobs. The wrapper is portable to a number of systems thanks to the use of standards such as MPI and pthreads as well as through compatibility features such as the process pool and tunable I/O parameters. We are unaware of any other solution that offers these performance improvements in the form of a reusable software wrapper.

It is our hope that the tool will be used by researchers for running existing programs. Perhaps more importantly, we hope the tool can be used to help developers run new programs on HPC resources without the need to learn parallel programming or invest time on researching optimizations. Developing software for distributed systems can be overwhelming, but HSP-Wrap can help quickly enable a subset of data-intensive applications to scale comfortably on petascale systems.

References

[1] H. You, B. Rekapalli, Q. Liu, and S. Moore, “Autotuned parallel I/O for highly scalable biosequence analysis,” in *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*. ACM, 2011, p. 29.

[2] B. Rekepalli and A. Vose, “Petascale genomic sequence search,” in *Proceedings of The 11th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, 2011.

[3] B. Rekepalli, A. Vose, and P. Giblock, “HSPp-BLAST: Highly scalable parallel PSI-BLAST for very large-scale sequence searches,” in *3rd International Conference on Bioinformatics and Computational Biology (BICoB)*, 2012, pp. 37–42.

[4] E. Check Hayden, “Technology: The $1,000 genome,” *Nature*, vol. 507, pp. 294–295, March 2014.

[5] S. D. Kahn, “On the future of genomic data,” *Science(Washington)*, vol. 331, no. 6018, pp. 728–729, 2011.

[6] T. Attwood, A. Gisel, N. Eriksson, and E. Bongcam-Rudloff, “Concepts, historical milestones and the central place of bioinformatics in modern biology: a European perspective,” *Bioinformatics-Trends and Methodologies*, 2011.

[7] R. Stevens, C. Goble, P. Baker, and A. Brass, “A classification of tasks in bioinformatics,” *Bioinformatics*, vol. 17, no. 2, pp. 180–188, 2001.

[8] D. J. Lipman and W. R. Pearson, “Rapid and sensitive protein similarity searches,” *Science*, vol. 227, no. 4693, pp. 1435–1441, 1985.

[9] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, “Basic local alignment search tool,” *Journal of molecular biology*, vol. 215, no. 3, pp. 403–410, 1990.

[10] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman, “Gapped BLAST and PSI-BLAST: a new generation of protein database search programs,” *Nucleic acids research*, vol. 25, no. 17, pp. 3389–3402, 1997.

[11] S. R. Eddy, “Profile hidden Markov models.” *Bioinformatics*, vol. 14, no. 9, pp. 755–763, 1998.

[12] R. C. Edgar, “MUSCLE: multiple sequence alignment with high accuracy and high throughput,” *Nucleic acids research*, vol. 32, no. 5, pp. 1792–1797, 2004.

[13] ——, “MUSCLE: a multiple sequence alignment method with reduced time and space complexity,” *BMC bioinformatics*, vol. 5, no. 1, p. 113, 2004.

[14] B. K. Shoichet, "Drug discovery: Nature's pieces," *Nature chemistry*, vol. 5, no. 1, pp. 9–10, 2013.

[15] H. M. Berman, "The protein data bank: a historical perspective," *Acta Crystallographica Section A: Foundations of Crystallography*, vol. 64, no. 1, pp. 88–95, 2007.

[16] I. D. Kuntz, J. M. Blaney, S. J. Oatley, R. Langridge, and T. E. Ferrin, "A geometric approach to macromolecule-ligand interactions," *Journal of molecular biology*, vol. 161, no. 2, pp. 269–288, 1982.

[17] P. T. Lang, S. R. Brozell, S. Mukherjee, E. F. Pettersen, E. C. Meng, V. Thomas, R. C. Rizzo, D. A. Case, T. L. James, and I. D. Kuntz, "DOCK 6: combining techniques to model RNA–small molecule complexes," *RNA*, vol. 15, no. 6, pp. 1219–1230, 2009.

[18] A. R. Hoffman and J. F. Traub, *Supercomputers: directions in technology and applications.* National Academies Press, 1989.

[19] H. Fujii, Y. Yasuda, H. Akashi, Y. Inagami, M. Koga, O. Ishihara, M. Kashiyama, H. Wada, and T. Sumimoto, "Architecture and performance of the Hitachi SR2201 massively parallel processor system," in *Parallel Processing Symposium, 1997. Proceedings., 11th International.* IEEE, 1997, pp. 233–241.

[20] D. A. Reed, *Scalable Input/Output: achieving system balance.* MIT Press, 2004.

[21] N. Adiga, M. Blumrich, D. Chen, P. Coteus, A. Gara, M. Giampapa, P. Heidelberger, S. Singh, B. Steinmacher-Burow, T. Takken, M. Tsao, and P. Vranas, "Blue Gene/L torus interconnection network," *IBM Journal of Research and Development*, vol. 49, no. 2.3, pp. 265–276, March 2005.

[22] J. Lubchenco and J. Hayes, "A better eye on the storm," *Scientific American*, vol. 306, no. 5, pp. 68–73, 2012.

[23] D. D. Mishra, C. Murthy, K. Bhatt, A. Bhattacharjee, and R. Mundada, "Development and performance analysis of HPC based framework for cryptanalytic attacks," in *Proceedings of the CUBE International Information Technology Conference.* ACM, 2012, pp. 789–794.

[24] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson *et al.*, "XSEDE: accelerating scientific discovery," *Computing in Science & Engineering*, vol. 16, no. 5, pp. 62–74, 2014.

[25] NICS. Kraken – National Institute for Computational Sciences. [Online]. Available: http://www.nics.tennessee.edu/computing-resources/kraken

[26] TOP500. (2010) Kraken XT5 - Cray XT5-HE Opteron 6-core 2.6 GHz — TOP500 supercomputer sites. [Online]. Available: http://www.top500.org/system/176545

[27] The University of Tennessee. Newton HPC Program – high performance computing. [Online]. Available: https://newton.utk.edu

[28] B. Leback, D. Miles, and M. Wolfe, "Tesla vs. Xeon Phi vs. Radeon a compiler writers perspective," in *CUG 2013 Proceedings*. Cray User Group, 2013.

[29] R. Brook, A. Heinecke, A. Costa, P. Peltz, V. Betro, T. Baer, M. Bader, and P. Dubey, "Beacon: Exploring the deployment and application of Intel Xeon Phi coprocessors for scientific computing," *Computing in Science & Engineering*, vol. 17, no. 2, pp. 65–72, March 2015.

[30] NICS. Beacon – National Institute for Computational Sciences. [Online]. Available: http://www.nics.tennessee.edu/beacon

[31] Sun Microsystems. (2011) Lustre 1.8 operations manual. [Online]. Available: http://docs.oracle.com/cd/E19495-01/821-0035-12/821-0035-12.pdf

[32] A. E. Darling, L. Carey, and W. Feng, "The design, implementation, and evaluation of mpiBLAST," in *Proceedings of ClusterWorld 2003*, 2003.

[33] H. Lin, X. Ma, P. Chandramohan, A. Geist, and N. Samatova, "Efficient data access for parallel BLAST," in *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*. IEEE, 2005, pp. 72b–72b.

[34] B. Rekapalli, C. Halloy, and I. B. Zhulin, "HSP-HMMER: a tool for protein domain identification on a large scale," in *Proceedings of the 2009 ACM symposium on Applied Computing*. ACM, 2009, pp. 766–770.

[35] Institute for Advanced Simulation. SIONlib: scalable massively parallel I/O to task-local files. [Online]. Available: http://www.fz-juelich.de/jsc/sionlib/

[36] C. Jin, S. Klasky, S. Hodson, W. Yu, J. Lofstead, H. Abbasi, K. Schwan, M. Wolf, M. Parashar, C. Docan *et al.*, "Adaptive IO system (ADIOS)," in *CUG 2008 Proceedings*, 2008.

[37] National Science Foundation. (2009) Data-intensive computing program. [Online]. Available: http://www.nsf.gov/funding/pgm_summ.jsp?pims_id=503324

[38] Cray Inc, "Using CrayPat," in *Optimizing applications on the Cray X1™ system*. Cray Inc, 2003, ch. 2.2.

[39] L. D. Crosby, R. G. Brook, B. Rekepalli, M. Sekachev, A. Vose, and K. Wong, "A pragmatic approach to improving the large-scale parallel I/O performance of scientific applications," in *CUG 2011 Proceedings.* Cray User Group, 2011.

[40] D. A. Wheeler. (2004) SLOCCount. [Online]. Available: http://www.dwheeler.com/sloccount

[41] M. Szeredi. (2013) FUSE: Filesystem in userspace. [Online]. Available: http://fuse.sourceforge.net

[42] Open MPI. (2015) FAQ: Tuning the run-time characteristics of MPI OpenFabrics communications (InfiniBand, RoCE and iWARP). [Online]. Available: http://www.open-mpi.org/faq/?category=openfabrics#ofa-fork

[43] ——. (2015) FAQ: General run-time tuning. [Online]. Available: http://www.open-mpi.org/faq/?category=tuning#fork-warning

[44] P. J. Braam and R. Zahir, "Lustre: a scalable, high performance file system," *Cluster File Systems, Inc.*, 2002.

Vita

Paul Giblock was born on April 20, 1984 in Freehold, New Jersey, and moved to Knoxville, Tennessee at the age of 4. Paul began programming computers when he was 11 years old, and received his Cisco Certified Network Associate (CCNA) certification while attending Farragut High School. After graduating in 2002, he began his undergraduate studies at East Tennessee State University in Johnson City. He graduated from college in 2006 with a bachelor degree in Computer Science, and moved back to Knoxville where he began his professional career as a software developer on various web applications.

Paul decided to work towards his master's degree in 2011. He left the commercial industry, and enrolled with The University of Tennessee's Department of Electrical Engineering and Computer Science as a graduate research assistant. It was during this period that he scaled the performance of informatics software at the National Institute for Computational Sciences (NICS) in Oak Ridge. Paul left NICS in 2013 to work full-time at Cisco Systems; where he is currently employed.