Concurrency Optimization for Integrative Network Analysis

R. Otto Barnes II

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

In

Computer Engineering

2013 MAY 3

Arlington, Virginia

Keywords: BMRF, simulated annealing, subnetwork identification, concurrency, parallelism

Concurrency Optimizations for Integrative Network Analysis

R. Otto Barnes II

Abstract

Virginia Tech's Computational Bioinformatics and Bio-imaging Laboratory (CBIL) is exploring integrative network analysis techniques to identify subnetworks or genetic pathways that contribute to various cancers. Chen et. al. developed a bagging Markov random field (BMRF)-based approach which examines gene expression data with prior biological information to reliably identify significant genes and proteins. Using random resampling with replacement (bootstrapping or bagging) is essential to confident results but is computationally demanding as multiple iterations of the network identification (by simulated annealing) is required. The MATLAB implementation is computationally demanding, employs limited concurrency, and thus time prohibitive. Using strong software development discipline we optimize BMRF using algorithmic, compiler, and concurrency techniques (including Nvidia GPUs) to alleviate the wall clock time needed for analysis of large-scale genomic data. Particularly, we decompose the BMRF algorithm into functional blocks, implement the algorithm in C/C++ and further explore the C/C++ implementation with concurrency optimization. Experiments are conducted with simulation and real data to demonstrate that a significant speedup of BMRF can be achieved by exploiting concurrency opportunities. We believe that the experience gained by this research shall help pave the way for us to develop computationally efficient algorithms leveraging concurrency, enabling researchers to efficiently analyze larger-scale data sets essential for furthering cancer research.

Acknowledgements

I would like to thank and acknowledge those who helped me in this journey. My committee chair and advisor, Dr. Jason Xuan for continued motivation, advice, and wisdom that seems to always keep me moving forward. Dr. Joseph Wang for insisting that "I'm the boss" and holding true to that nugget of wisdom. Dr. Ron Kriz for constant reminders and examples of communicating clearer and thinking interdisciplinary. Li Chen for the BMRF algorithm and MATLAB implementation. The CBIL team for zero complaints despite my constant questions. My employer for financing this academic endeavor and thanks to three particular co-workers for continuous motivation and improving my critical thinking: Ambrose Lewis, Phil Tamburello, and the late Steve Clark. Thank you to the developers of Google, Boost, Eclipse, BitBucket, Git, Spotify, Nvidia CUDA, Linux, Threadpool, Tiddlywiki, and the open source community for the tools to make this effort possible. Thanks to my friends for being good friends and thanks to Mom and Dad whose support and nurturing have always been unwavering.

R. Otto Barnes II

May 2013

Contents

Chapter 1

1. Introduction

1.1 Statement

One of the research efforts at the Virginia Tech Computational Bioinformatics and Bio-imaging Laboratory (CBIL) is focused on uncovering signaling pathways associated with estrogen receptor-positive (ER+) breast cancer. In the pursuit of which, a novel computational approach [Chen2012] has been developed for identifying subnetworks from gene expression data and protein-protein interaction (PPI) data; the approach is named bagging Markov random field (BMRF). The algorithm was developed and initially implemented in MATLAB and would benefit from optimization.

Concurrency analysis of BMRF can reveal optimizations that, given parallel capabilities of modern processor architectures, will achieve performance enhancements minimizing its execution time and enabling future efforts deemed intractable today. By applying industry standards and practices, we will convert or port the MATLAB implementation to C/C++ (a computer language with increased verbosity, flexibility, and performance) and leverage parallelism exploiting concurrency opportunities. Using Amdahl's law as a guide, we will evaluate the performance against predictions to identify optimal implementations on target hardware.

1.2 Organization

The remainder of Chapter 1 delves into the background of computational biology and navigates a path to BMRF and the necessary tools / techniques for making performance improvements. Chapter 2 dissects BMRF into its basic components, discuss how it was implemented in C/C++, and illustrate the process for optimization. Chapter 3 focuses on experimentation, performance results, and performance analysis. Chapter 4 summarizes findings and provides avenues for future work.

References are identified by first author's last name, followed by year published with no spaces and surrounded by brackets, e.g.: [Barnes2013]. The bibliography will list the full reference organized in alphabetical order by reference identifier. When referencing computer code or commands, we will use a fixed typeface, e.g.: `intersect` would reference a function that would perform a set intersection.

1.3 Computational Biology

Computational biology is a field at the crossroad between biological discovery and computational science. Biologists, chemists, and medical researchers explore the molecular mechanism of human organs to understand the cause of disease and cancer. Electrical / computer engineers and computer scientists develop engineering techniques, computer software, and computational algorithms to analyze biological data and model biological systems. Each discipline is focused on separate but complementary goals for cancer research. Under computational biology, understanding of biological data is aided with computational expertise in handling and analyzing enormous amounts of genomic data accumulated. Beginning in the 1970s under the name bioinformatics, the field has grown in breadth and depth. For example, protein folding was found to be a computationally intractable problem [Berger1998] and now molecular classification of cancer is a sub-area of computational biology [Yakhini2011].

The Human Genome Project formally began in 1990 to identify and sequence all human genes. A monumental task that in 2003 was completed two years ahead of schedule due to advancements in sequencing technology [Buris2011]. As Wetterstrand points out, the National Institute of Health's (NIH) National Human Genome Research Institute (NHGRI) Genome Sequencing Program (GSP) has tracked the cost of sequencing since 2001 [Wetterstrand2013]. Illustrated by Figure 1, it is clear that genome sequencing is out pacing Moore's Law.

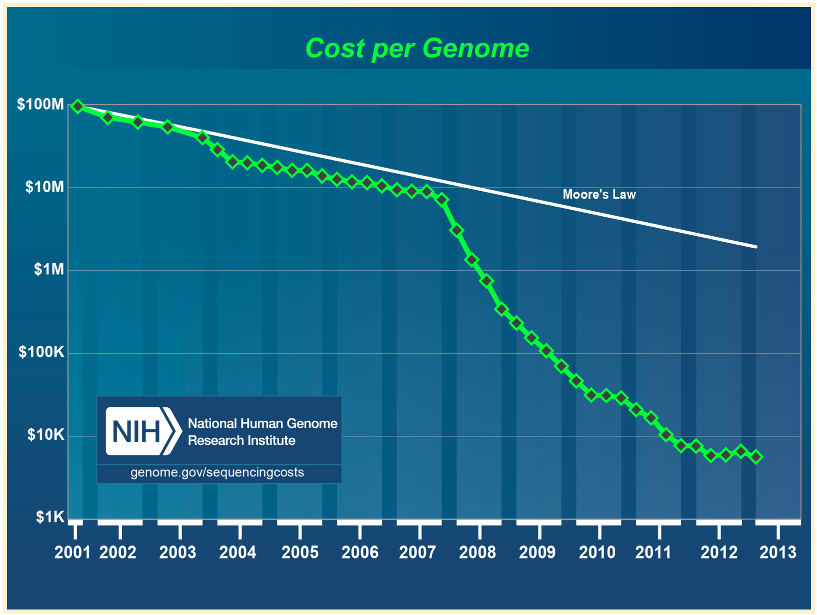


Figure 1: Cost to sequence per Genome as tracked by NIH's NHGRI GSP with a hypothetical Moore's Law trend overlaid. [Wetterstrand2013]

In the perspective of history, the latest software crisis that Dr. Amarasinghe cites in his Multicore Programming Primer Course has been upon us for almost a decade. The 60's and 70's were spent dealing with the non-portable, slow, and complex assembly language programming model, which led to, in the 80's and 90's, the invention of C and consequently the maintenance nightmare resulting from a lack of reuse [Amarasinghe2007]. Today, we are faced with the end of the *perceived* Moore's law where microprocessors are not gaining in clock speed by a factor of two every two years. Instead, what Gordon Moore predicted in 1965 was that the number of components per integrated function or number of transistors per unit area would double every year [Moore1965]. The *preceived* version held true for a few decades but was the by-product of the transistor shrinking. There was a concern with leakage current and power required to continue the doubling of clock frequency due to power and thermal egress concerns [Kim2003]. Over the past decade, the increased number of transistors per silicon wafer has led to the multi-core and parallelism software crisis. As, biological data is becoming exponentially cheap and abundant, and computational hardware is exponentially becoming more powerful, algorithms must be developed with parallelism at their forefront to remain useable year over year.

1.4 Biological Data Sources

The central dogma of molecular biology describes the interactions between DNA, RNA and proteins. Francis Crick described their relationship in 1958 and then again in 1970 where he updated to include special transfers (dotted arrows) [Crick1970].

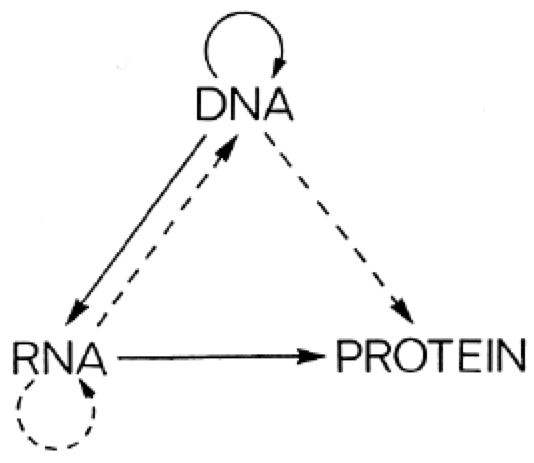


Figure 2: Crick's tentative classification of the central dogma: solid arrows show general transfers; dotted arrows show special transfers, absent arrows are undetected transfers.

In Figure 2, we can see that DNA is transcribed to mRNA which is then translated to proteins. Because of the complexity of biological systems, it is essential to use multiple data sets to understand the unhealthy change in human biology [Hanash2004]. We will work with two types of biological data which cover all three of these molecular structures: gene expression data and Protein-Protein Interaction (PPI) data.

Gene expression data are acquired from microarray technique such as Affymetrix GeneChip Arrays, which is a technology that relies on hybridization between two complementary DNA strands (one half of the double-helix). The basic idea of microarray technique is to measure the concentration of certain molecular sequence by utilizing the hybridization between the signature sequence of this molecular sequence and another complementary DNA sequences [Allison2006] [Hoheisel2006]. The commonly used microarray chip consists of a series of microscopic spots of DNA oligonucleotides, each of which contains specific DNA sequence known as probe (or reporter) to hybridize a corresponding cDNA or cRNA sample (also known as target) under well controlled experimental conditions. Tens of thousands of probes are laid out in a hard surface such as glass and silicon using surface engineering. Using these probes, a microarray chip measures the activities of huge amounts of genetic targets in parallel.

Proteins are plentiful and diverse in the human body. They are responsible for transporting and storing critical substances, a building block of cells, and their interaction with each other have varied but significant biological functionality. Protein, the end product of gene transcription and translation processes, is responsible for every biological process and molecular function of living cells. The defects of proteins can lead to diseases such as muscular dystrophy [Khurana2003], which is single gene mutation disease, and the rewiring of the protein signaling network could contribute to the metastasis mechanisms [Chuang2007] and clinical outcome [Taylor2009] of breast cancer. Therefore, it would be very important to understand the function of protein and the biological relationship among proteins. With the advent of yeast2hybrid technique, protein-protein interactions (PPIs) can now be measured in a high-throughput way and protein-protein interaction data are readily available for several species. PPI data can be organized in an undirected graph. Each protein receives an identifier and if an interaction occurs then that interaction is logged in a dataset consisting of two columns, one for each protein identifier. Two such databases are [Mishra2006] and [Mering2003].

1.5 Subnetwork Analysis

Examining yeast, Idecker et al. investigated protein-protein and protein→DNA interaction data along with mRNA expression data to identify active subnetworks. Active subnetworks are connected regions that show significant changes in expression over varying conditions. They developed a method of ranking the biological activity of a particular subnetwork using a novel scoring they call z-score [Ideker2002]. They demonstrated detection of subnetworks consistent with known regulatory circuits (and expert knowledge from yeast biology) using Simulated Annealing (SA) approach from [Kirkpatrick1983].

The SA algorithm follows Boltzmann factor such that the number of entropy is highest at high temperature and as the temperature decreases, less and less possible lower energy states exist. Nodes are randomly added or removed based on a decreasing temperature or cooling schedule which enables the algorithm to hop from an unacceptable local minimum and arrive at the global optimum [Duda2001].

Chuang et al. applied the z-score method to classify metastatic vs. non-metastatic tumors [Chuang2007]. Dittrich applies a novel technique to achieve an optimal solution based on a mixture model for signal-noise decomposition and linear programming techniques [Dittrich2008]. Chen et al. present a novel subnetwork identification algorithm using Idecker z-scores, SA with a Markov random field (MRF) maximum a posterior (MAP) estimator, Dittrich's false-discovery rate for confident subnetworks [Chen2012]. A Markov random field (MRF) is a set of stochastic or random variables whose relationships, conditional dependence, can be described as a undirected graph. This is an apt model for describing the association of genes and proteins as a sub-network. Additionally, Chen et al. combine bagging or bootstrap aggregation to minimize the effect of gene expression measurement noise as demonstrated in [Simoes2012]. The algorithm is consequently named Bagging Markov Random Field (BMRF).

Introduced by [Breiman1996], bagging consists of resampling a dataset with reuse or replacement such that a classifier is tested against multiple versions of a dataset to overcome overfit to ensure stability [Duda2001]. Breiman cautions that bagging cannot overcome all instability.

We provide a detailed dissection of BMRF in Chapter 2.

1.6 Parallelism Techniques

The effort of parallelizing, most basically, is the decomposition of an algorithm into subproblems that can safely be executed at the same time. The algorithm's structure as a whole or in parts can be summarized by a variety of patterns for finding concurrency but the goal is to exploit as much concurrently allowed the target hardware by identify tasks and their data dependencies [Mattson2005].

Amdahl's law describes the expected performance improvement when a portion of a program is parallelized [Amdahl1967]. As an example, if a half a program can be optimized by using two processors then 50% of the program execution can receive a 2x speedup for a total execution time of 75% of the original. We will use Amdahl's law in estimating the speedup factor or gain for BMRF when we employ parallelism optimizations.

Artful engineering of a solution must be made according to a variety of factors not necessarily coupled directly with the algorithm. In software design speak these are star-abilities or *abilities such as: usability, readability, extensibility, and maintainability. Often to aid in decision making we employ the Pareto principle or 80/20 rule and Amdahl's Law which guides us to where our effort should be spent on parallelism and the minimum runtime expected.

The parallelism software crisis has produced a variety of tools to date that are at our disposal: from microprocessor parallel instruction sets, compiler supported parallel directives, compiler optimizations, parallel array processors, to commodity personal computer clusters or cloud technologies.

In 2013, there is a high probability that algorithms will be developed in a high-level language such as Mathwork's MATLAB on a commodity personal computer equipped with a x86 instruction set microprocessor. This is the platform that has evolved through the combination of Moore's Law and the software crises of the previous five decades and has resulted in low cost and highly replicable computing environments. Thus, our design for usability concerns is somewhat favors this paradigm. Flynn's Taxonomy illustrates this constraint as SISD is the defacto standard and our options of parallelism are most easily attained by data parallelism or SIMD [Flynn1972], Figure 3.

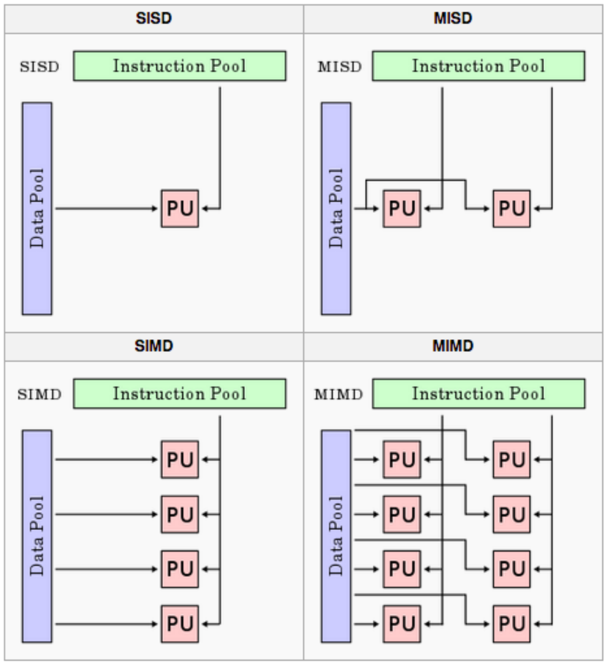


Figure 3: Illustration of Flynn's Taxonomy. http://en.wikipedia.org/wiki/Flynn's_taxonomy

Compiler optimizations and microprocessor streaming SIMD extensions (SSE) go hand in hand. The Intel Compiler and GCC support multiple extensions (MMX, SSE, SSE2/3/4/4.2) that enable software to operating the same instruction simultaneously on multiple words [Intel2011]. Each processor supports a range of these extensions and some compilers can target code for those specific capabilities. Often, mathematical libraries like LAPACK, BLAS, VSIPL will be compiled with these streaming SIMD extensions enabled.

OpenMP is a specification like defines compiler directives to enable development of portable parallel programs. The developer must explicitly define parallel regions of code with the use of #pragma statements making sure to respect private and shared data variables with in these regions. The specification does not make claims to compiler implementations checking data dependences, conflicts, race conditions, or deadlocks

[OpenMP2008].

Graphical Processing Units (GPUs) are special purpose SIMD processors. One of the demanding tasks in a Computer Graphics pipeline is to shade the three-dimensional scene. Shading requires ray tracing and involves multiple light sources, objects, reflectivities, and many other optical details to be calculated for every pixel in the frame buffer. Ray tracing is inherently parallel because photons do not interact with each other. The demand of the computer gaming industry has led to the some acceleration over Moore's law; the latest Nvidia GPU boasts 7.1 billion transistors while Intel is in the low 3 billions [Nvidia2012]. Figure 4 shows the peak theoretical floating point operations per second (FLOPS) of Nvidia Tesla line of GPUs released over roughly four years.

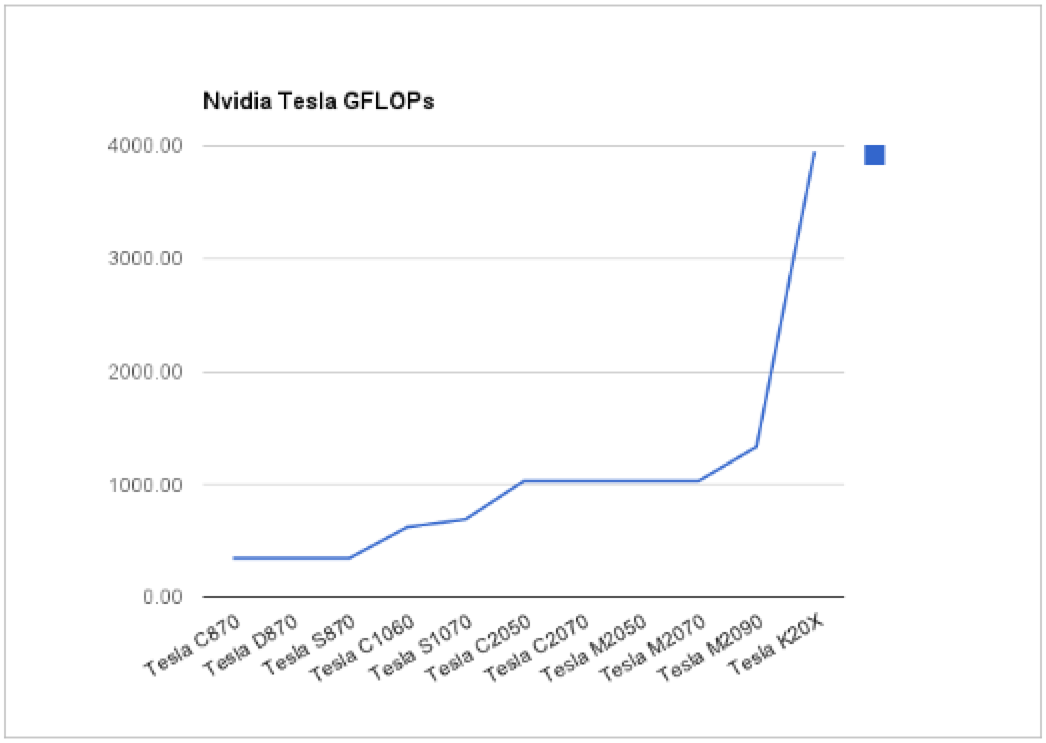


Figure 4: Peak theoretical FLOPS per second of Nvidia's Tesla GPU releases over 2008-2012.

Given this massive quantity of computation, Ian Buck et. al developed a streaming or data parallel language with GPU support at Stanford [Buck2004]. This General Programming on GPUs (GPGPU) later became a fully supported Nvidia product called CUDA [Dove2006]. Its important to note that CUDA exposes the GPU as a Single Program Multiple Data (SPMD) paradigm rather than a pedagogical SIMD processor. The result is a very capable processor with ever increasing computational capability.

Chapter 2

2. Design and Implementation

2.1 Algorithm Description

Bagging Markov random field (BMRF) as developed by Chen et al. is a novel subnetwork identification algorithm using [Idecker2002] z-score calculation, Simulated Annealing [Kirkpatrick1983] with a Markov random field maximum a posterior estimator (MRF-MAP) that is bagged or repeated with resampling and combines [Dittrich2008] false-discovery rate against a null-hypothesis for confident subnetworks. Figure 5, illustrates this process.

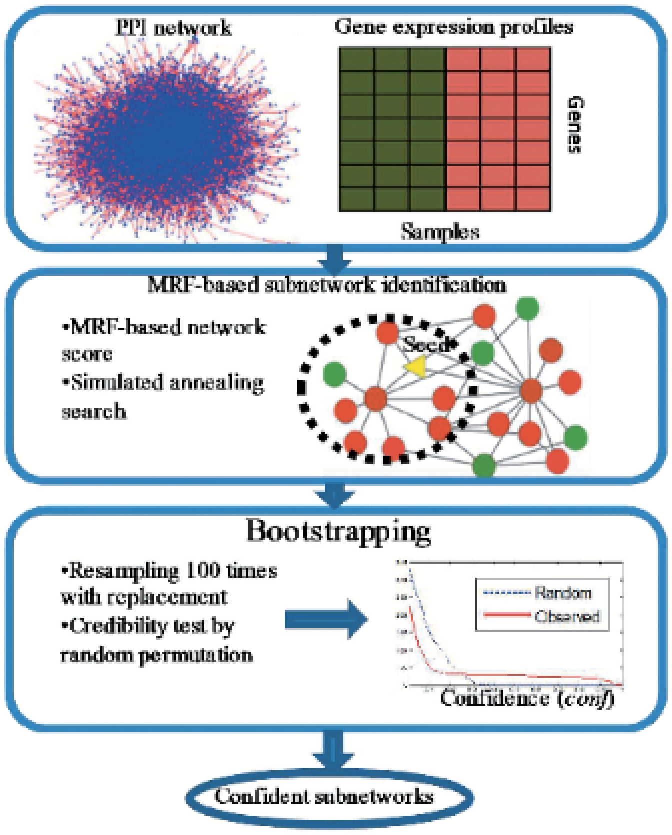


Figure 5: Overview of BMRF subnetwork identification form microarray gene expression profiles and PPI network.

Idecker et al. provide a robust method to normalize gene expression data to standard normal [Idecker2002]. They define a bivariate normal probability density function, use the standard student's t cdf to compute p-values, apply bounds corrections, then take the inverse normal cdf of the p-values:

$$z_i = \Phi^{-1}(1 - p_i)$$

And, calibrate each score against the k number of z_i.

$$z_{n_i} = \frac{(z_i - \mu_k)}{\sigma_k}$$

The normalized z-score, z_{n_i}, is used and referred in BMRF as the gene score or just z-score. These z-scores are used by Stochastic Simulated Annealing (SA), an iterative process for randomly adding or removing nodes to optimize a subnetwork score where the probability of adding a node diminishes (line 9 of Listing 1) as the temperature decreases over each iteration.

```
1. begin initialize T(k), k_max, s_i(1), w_ij for i, j - 1,...,N
2.       k ← 0
3.       do k ← k+1
4.          do select node i randomly; suppose its state is S_i
5.             E_a ← $-1/2\sum_{j}^{N_i} w_{ij}s_is_j$
6.             E_b ← -E_a
7.             if E_b < E_a
8.                then s_i ← -s_i
9.                else if $e^{-(E_b-E_a)/T(k)}$ > Rand[0,1)
10.                  then s_i ← -s_i
11.         until all nodes polled several times
12.      until k=k_max or stopping criterion met
13.   return E, s_i, for i-1,...,N
14. end
```

Listing 1: Stochastic Simulated Annealing from [Duda2001]

Where E is replace by the value of the current subnetwork which is defined by MRF-MAP:

$$NetScore(G) = -U(f|Z) = \tfrac{1}{m}\sum f_i - \tfrac{\lambda}{k}\sum_{(i,i')\in E}\left(\frac{f_i}{\sqrt{d_i}} - \frac{f_{i'}}{\sqrt{d_{i'}}}\right)^2 - \tfrac{\gamma}{m}\sum_{i\in S}(Z_i - f_i)^2/2\,,$$

$$f = \left(\tfrac{2\lambda L}{k} + \tfrac{\gamma I}{m}\right)^{-1}\left(\tfrac{1+\gamma z}{m}\right),$$

$$\mathbf{L}(u,v) = \begin{cases} 1 & \textit{if } u = v \textit{ and } d_u \neq 0 \\ -1/\sqrt{d_u d_v} & \textit{if } u \textit{ and } v \textit{ are adjacent} \\ 0 & \textit{otherwise} \end{cases},$$

I is the identify matrix of L, z is the observed z-score, γ and λ are defined as 1.0.

Bagging is done with B=100 bootstraps where the gene expression data is resampled according to the early and late classes and the SA is performed and recorded. This bootstrapping is then repeated for the null-hypothesis, where randomization is performed without respect to classes.

Confidence Analysis then generates a false definition rate (FDR), and selects genes where the confidence is greater than 0.05. FDR is defined, for B bootstraps:

$$conf(gene_i) = \frac{1}{B} \sum_{b=1}^{B} f(gene_i^b),$$

$$FDR(conf_0) = \frac{\#\ of\ genes\ with\ conf_{baseline} \geq conf_0}{\#\ of\ genes\ with\ conf_{bootstraped} \geq conf_0}$$

2.2 MATLAB Refactor

We restructured the reference BMRF implementation for two key purposes: 1) isolating inputs and outputs so that validation and performance analysis could be performed unencumbered 2) grouping small collections of functions such that comparative testing becomes reasonable. We did the following:

- Separated simulated data generation from BMRF core m-files
- Grouped z-score calculation into `genescore.m`
- Grouped network candidate generation into `netcand.m`
- Added Cytoscape (cytoscape.org) generation code for graphical visualization of subnetworks with undirected connections.

- Parameterized inputs for easy selection of different datasets.

This effort requires very little knowledge of the algorithm but compels understanding by requiring a strict adherence to the algorithm's control logic. The high-level inputs / outputs structure then becomes Figure 6, where `demoMatlab`, `demoCpu`, etc... are MATLAB wrappers around variants of BMRF in other programming languages. The gene expression generation code is confined to generate.m and the Cytoscape file format generation code is confined to validate.m.

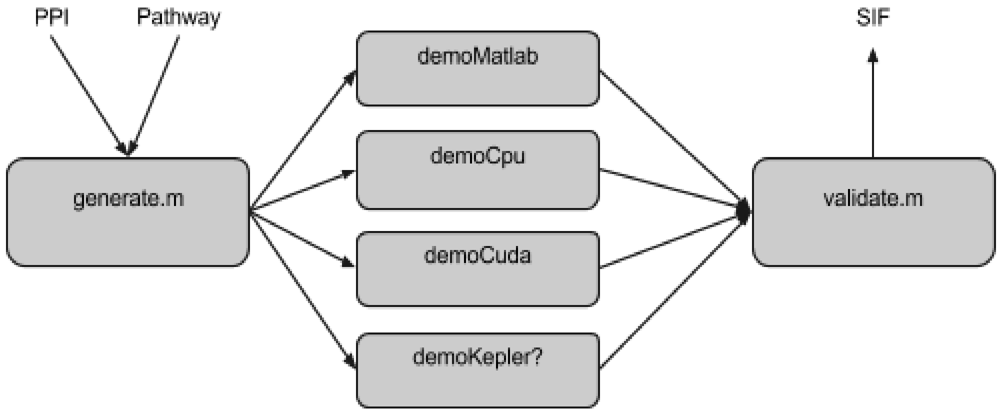


Figure 6: Refactored MATLAB high-level scripts controlling inputs / outputs for multiple implementations. Note the separation of generation and validation scripts.

From a software design viewpoint, this refactoring effort is necessary for any MATLAB migration to a different language and is even beneficial when evolving the algorithm and comparing versions. "Prefactoring uses insights you have gleaned from your experience as well as the experience of others in developing software" [Pugh2005]. We suggest algorithm developers heed this experience, regardless of their development environment.

The MATLAB script for executing the refactored matlab code is `demoMatlab`, dependency graph is shown in Figure 7. With this view of the algorithm, we also take note of matlab functions we will need to implement in C/C++ which are those within the blue box.

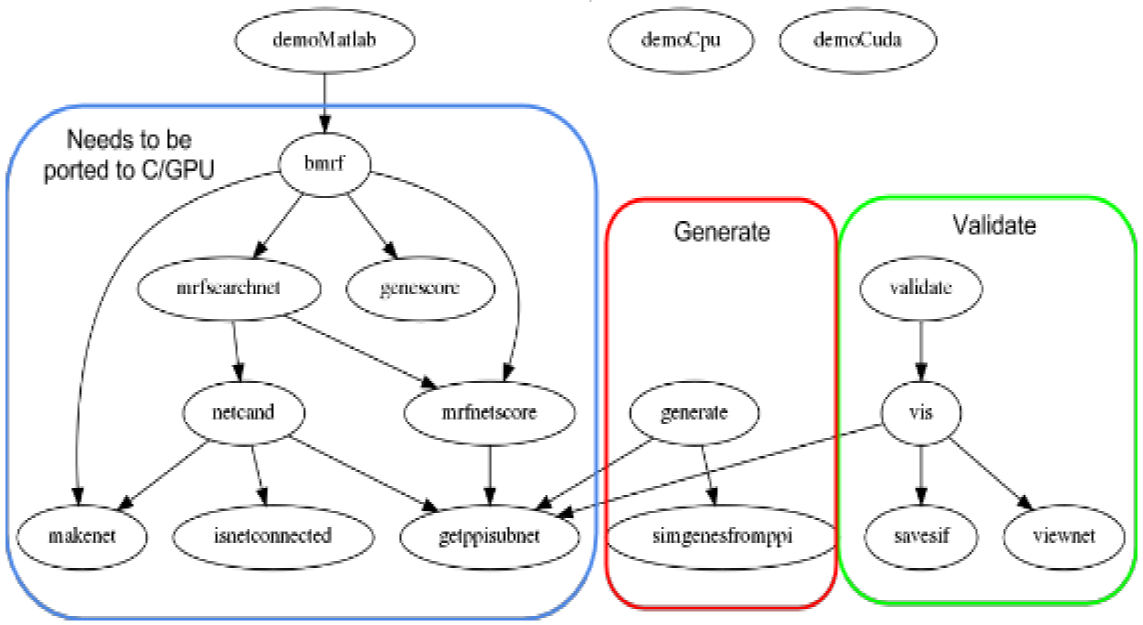


Figure 7: Refactored MATLAB showing the dependency graph between scripts.

2.3 C/C++ Implementation

While MATLAB provides a very convenient algorithm development environment, its many abstractions and interpreted nature make it perform non-optimally. We have selected a C implementation with as few as possible C++ features to be most compatible with Nvidia CUDA™ compiler which supports ANSI C for device code.

We use bitbucket.org, Git based and web accessible, for source control. Cmake for the build system as it offers a drastic simplification for describing the build rules of a project over the Make of 1970's. To aid in input argument parsing, we use docopt which compiles our text based usage statement into compilable C code. And use `std::rand` for our random number generation which is mostly used in the terms of true or false; we develop a wrapper that provides a probability adjustment based on the maximum generated random number.

To read and write MATLAB's .mat format, we set MATLAB's output format to version 7.3 and use Hierarchical Data Format version 5 (HDF5, hfgroup.org) library in our C/C++ implementation. The repository layout is available in Appendix A. We discovered that while MATLAB writes in HDF5 format, it cannot read HDF5 files outputted from C/C++. MATLAB requires a header that reads "`MATLAB 7.3 MAT-file, `" zero filled until the HDF5 header block. We used a facade design pattern to abstract away this incompatibility.

We define an application named `bmrf-cpu` to read input files select the appropriate implementation (we will have two) and execute BMRF. The `main` function performs this procedure by translating `DocoptArgs` into `BmrfParams` which is passed to the `BmrfHostAlgorithm` class. With the exception of the main and docopt related material, all code is placed in a library so that both the `bmrf-cpu` and `bmrfc-tests` (unit test driver application) can use common code. We follow test driven development pragmatically with a fairly complete set of unit tests for dependent functions. The GPU implementation is organized under the `BmrfDeviceAlgorithm` class. We define a base class `IBmrfAlgorithm` to provide a unified interface. Figure 8 presents a UML diagram for these relationships.

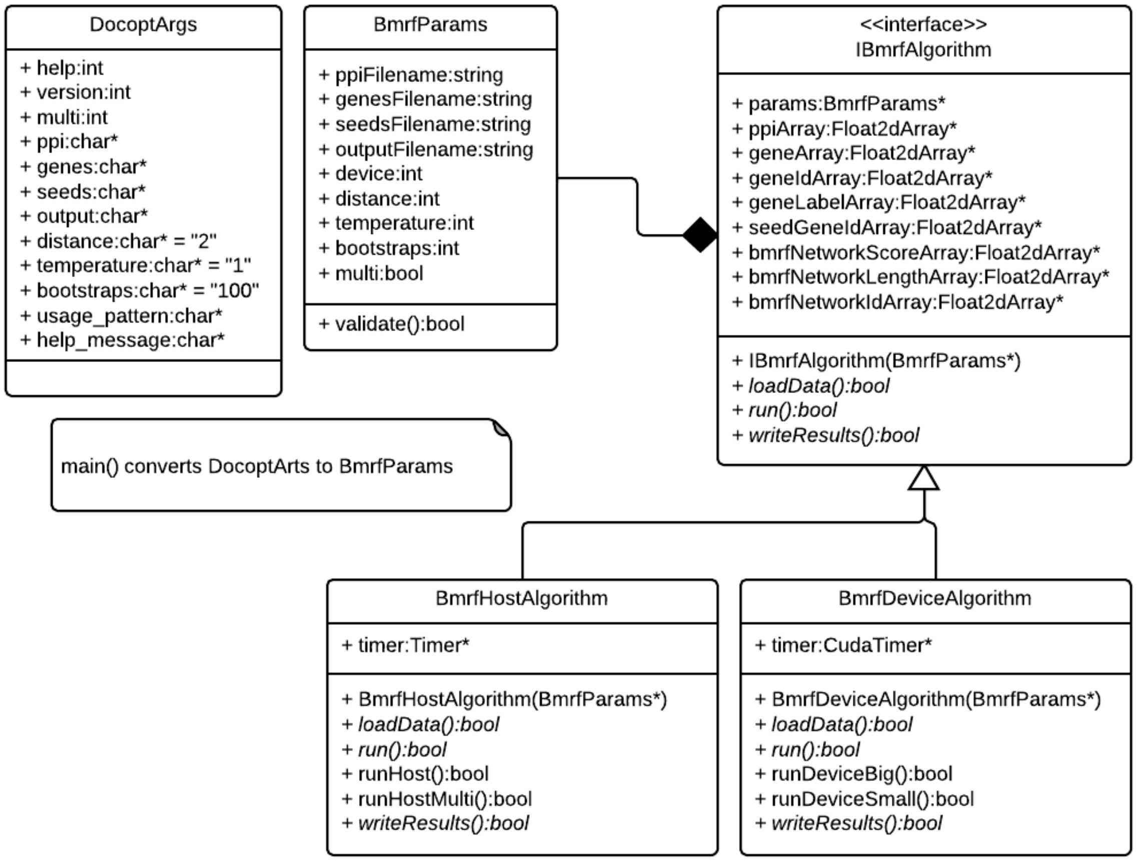


Figure 8: C/C++ implementation for BMRF

Data organization is critical to the performance of an application. We define two templates, one for one-dimensional structures (vectors) and one for two-dimensional structures (matrices) such that we may define typedefs for three intrinsic data types: unsigned int (shortened sometimes as uint), float, and double. This gives us six structures in the format <data type><# dimension>dArray, e.g: `Float2dArray`. While there are many vector and matrix classes, we were highly concerned with keeping them light weight to ensure quick migration to the GPU. Figure 9 depicts the UML for these data structures. We define basic accessor functions (x,y) for two-dimensional and [x] for one-dimensional structures, copy constructors, and accessors for the number of elements in the array.

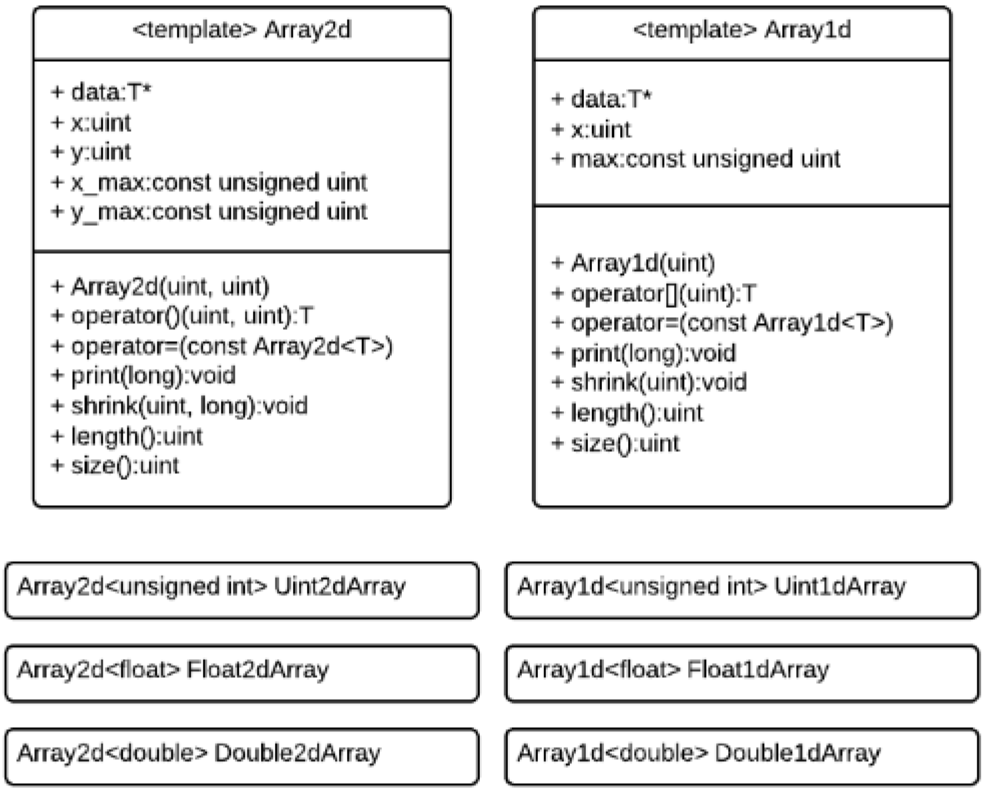

<template> Array2d
+ data:T*
+ x:uint
+ y:uint
+ x_max:const unsigned uint
+ y_max:const unsigned uint
+ Array2d(uint, uint)
+ operator()(uint, uint):T
+ operator=(const Array2d<T>)
+ print(long):void
+ shrink(uint, long):void
+ length():uint
+ size():uint
<template> Array1d
+ data:T*
+ x:uint
+ max:const unsigned uint
+ Array1d(uint)
+ operator[](uint):T
+ operator=(const Array1d<T>)
+ print(long):void
+ shrink(uint):void
+ length():uint
+ size():uint
Array2d<unsigned int> Uint2dArray
Array1d<unsigned int> Uint1dArray
Array2d<float> Float2dArray
Array1d<float> Float1dArray
Array2d<double> Double2dArray
Array1d<double> Double1dArray

Figure 9: Templated data structures for vector and matrices use throughout C/C++ implementation.

We described the objectives and algorithm design in previous subsections. This subsection follows the BMRF MATLAB implementation's dependency chart closely, Figure 7, in that each C/C++ function has been closely named to the script (.m or m-files) names. For instance, `mrfSearchNet` is a C/C++ function that parallels `mrfsearchnet.m`, and `netcand` parallels `netcand.m` functionality.

Figure 10, provides the UML flow diagram for executing `mrfSearchNet` for the number of bootstraps and the number of seed genes in search. The initial single threaded implementation, `BmrfHostAlgorithm::runHost`, is shown and closely resembles the BMRF MATLAB implementation. The bagging or bootstrap aggregation is fully visible here. Note that the number of baselines is the same as the number of bootstraps.

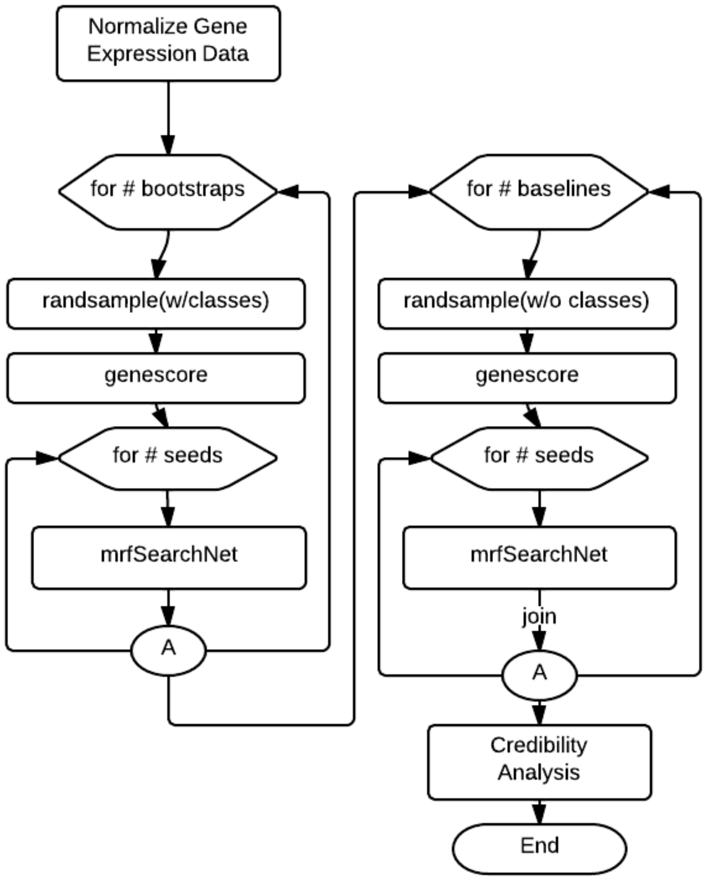


Figure 10: Flow diagram showing how `mrfSearchNet` is called in terms of bagging or bootstrap aggregation.

We will return to credibility analysis and gene score calculation. The core of the algorithm is `mrfSearchNet`, which is where the simulated annealing is performed. We use 1000 iterations [Chen2012] showed that a reduction from 2000 to 1000 was reasonable. Because simulated annealing has a decreasing temperature, the possibility of making changes to the subnetwork decreases to a level where only one or two modifications are made between iteration 500 and 1000. While developing this implementation, we temporarily terminated the SA when no modifications were made within 200 iterations; a shortcut which produced reasonable but sub-optimal results with significant wall clock time reduction. This greatly aided in implementation until other performance improvements could be added.

Most of the time spent in `mrfSearchNet` function is within `netcand` which generates the candidate networks based on the seed gene selected in `BmrfHostAlgorithm::runHost`. It is possible and in fact common for `netcand` to produce an empty set and thus must be tested for an empty set. Once a non-empty set is generated, it requires two intersections to derive a set with the new randomly selected gene from the candidate network. This new set determines if this iteration will add or remove an item from the subnetwork. The reader is encouraged to follow the flow diagram in Figure 11.

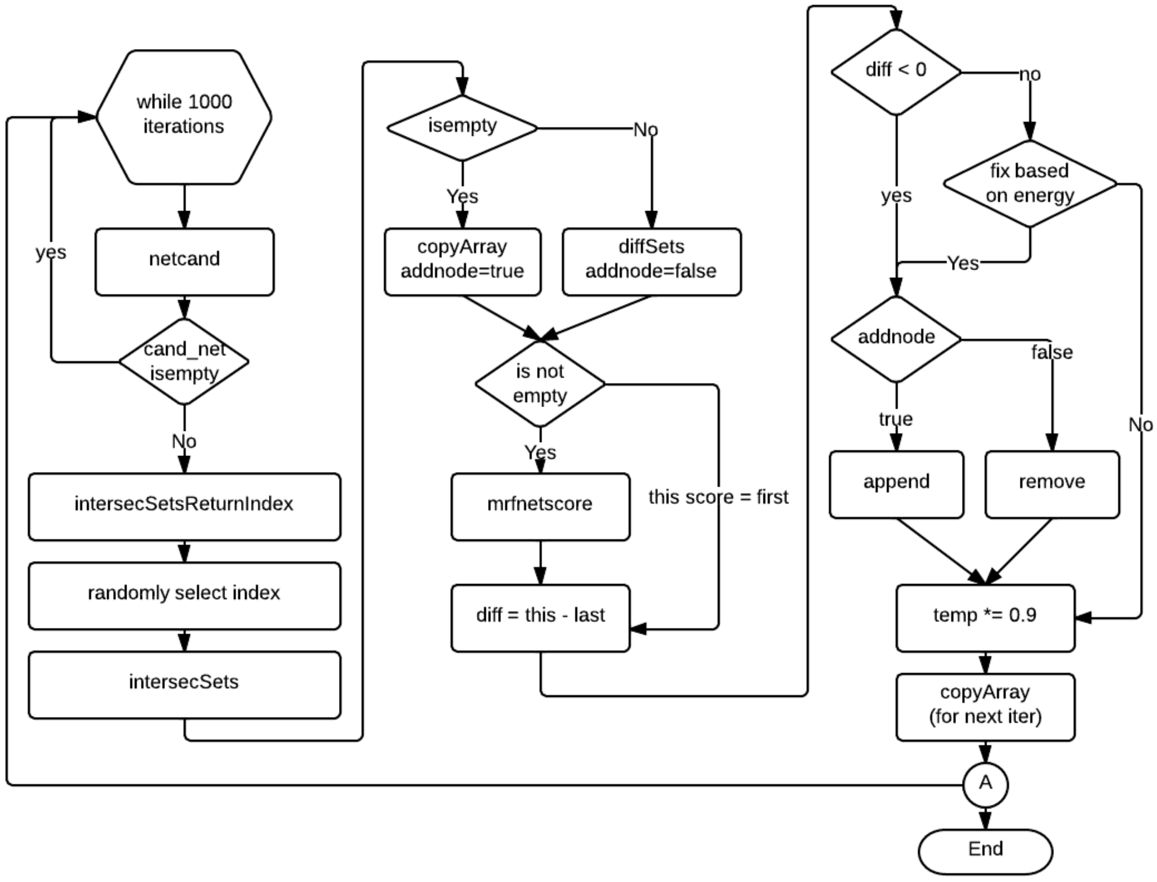


Figure 11: `mrfSearchNet` function's (simulated annealing) UML flow diagram.

We now calculate the score of the subnetwork (previous subnetwork with a node removed `if addnode == false`) or use the initial network score if all nodes have been removed. If the network score increased we add or remove the node based on a probability based on the difference in network score over the current system temperature. We then add or remove and prepare for the next iteration by decreasing the temperature and saving the results of this iteration.

$$p[i] = e^{-\frac{U(f[i])-U(f[i-1])}{T}}$$

We have to describe two sub-functions of `mrfSearchNet`, namely netcand and `mrfNetScore`. We refactored the reference MATLAB implementation to create `netcand` originally because the same code was repeated in the `bmrf.m` MATLAB script as well as part of `mrfSearchNet`. We realized, however, that the `netcand` calls near the normalization section of bmrf were superfluous and other CBIL members confirmed and this contributed to a small performance improvement in the MATLAB implementation. It was particularly helpful to have a `netcand` unit test function as there were concerns with compatibility of `std::rand` and MATLAB's randsample. That being said, `netcand` is rather straightforward; it randomly decides to return the outer layer of ids or a set of ids connected to the previous gene. It may return an empty set. The right side of Figure 12 illustrates the function flow to extract connected genes using the PPI for each id in the previous network, all ids with a distance > 2 are pruned before returning the resulting set.

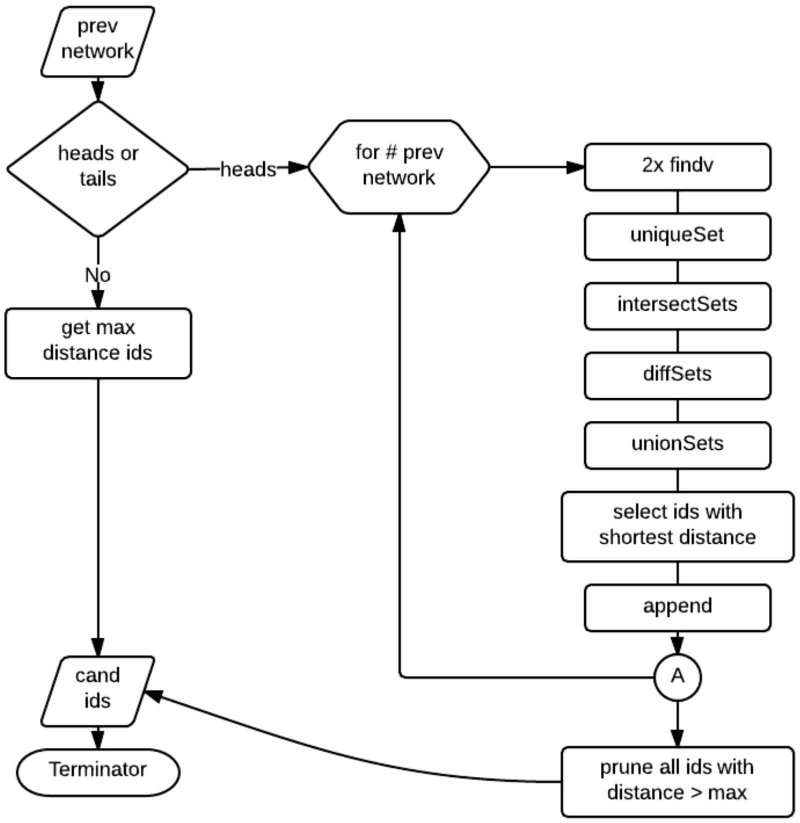


Figure 12: Candidate network generation function, `netcand`.

Moving to the second sub-function used in `mrfSearchNet`, `mrfNetScore` calculates the negative posterior potential function as defined in Chapter 2.1. Figure 13 shows the UML flow diagram.

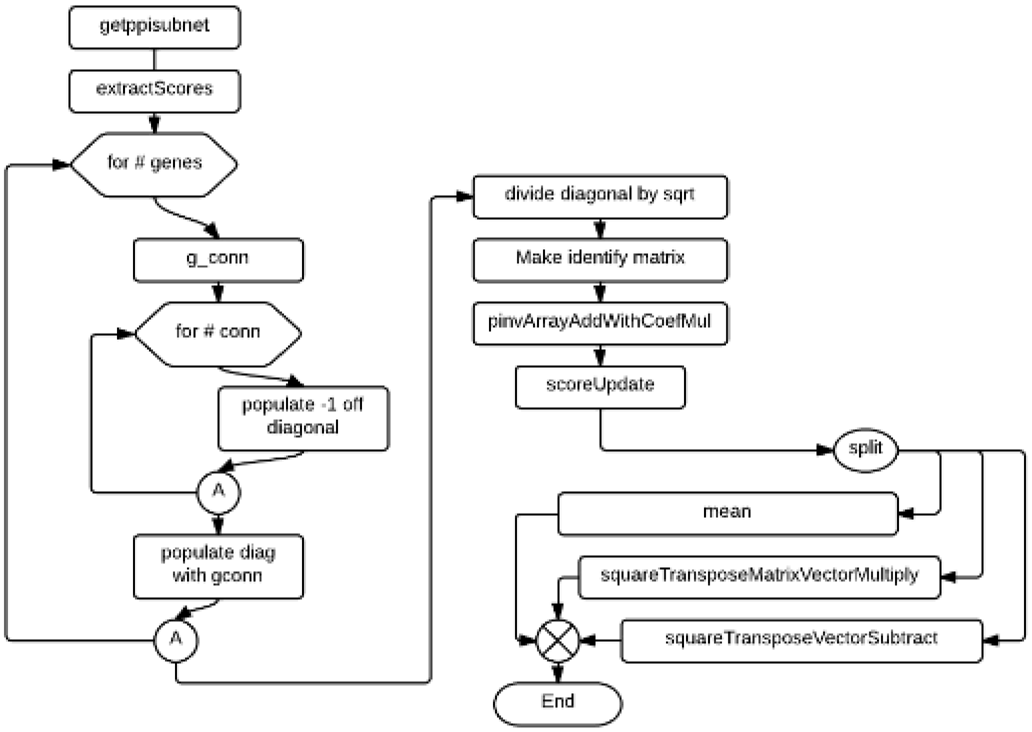


Figure 13: UML flow diagram for calculating negative posterior potential function (`mrfNetScore`).

There are a few functions in `mrfNetScore` that are rather straight forward but their names fail to fully capture their purpose:

- `extractScores` - given the subnetwork and supporting selection of PPI, return the matching z-scores. This function generates a mini z-score array.
- `g_conn` - given a gene id, extract a list of indices that map into the PPI array.
- `scoreUpdate` - helper function to perform matrix (NxM) vector (Nx1) multiplication where the vector, *v*, is scaled by $\frac{1+\gamma v}{N}$, gamma is 1.0.

A critical function is extracting the subnetwork given the protein-protein interaction array and gene ids. The PPI array is organized in two columns and N number of rows. For each row, the id in the first column is connected to the id in the second column and vise versa. Given a network of gene ids, the PPI subnetwork consists of all the fully connected gene ids, where both columns are represented in the gene id list or the intersection of search results from each column of PPI.

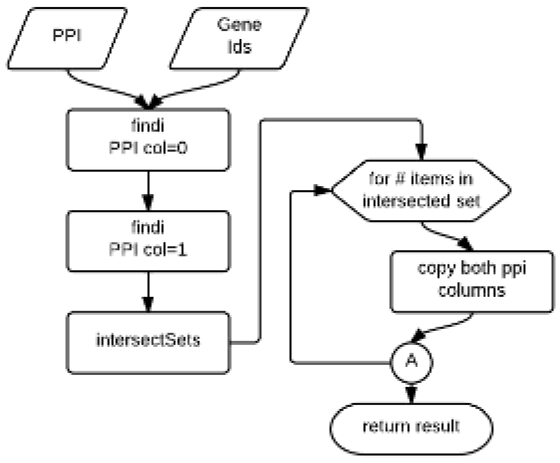


Figure 14: UML flow diagram for getppisubnet.

Lastly we will address `genescore` which is the generation of z-scores per [Idecker2002]. Which consists of calculating a normalized t-statistic, t-test cdf for p-values, bounds corrections, `icdf` to arrive at z-scores following a standard normal (illustrated on right side of Figure 15).

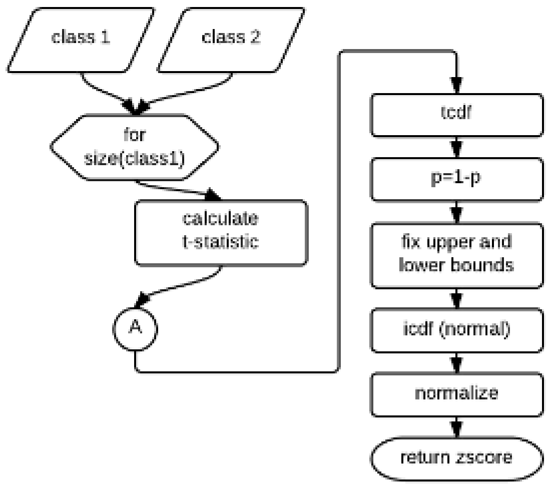
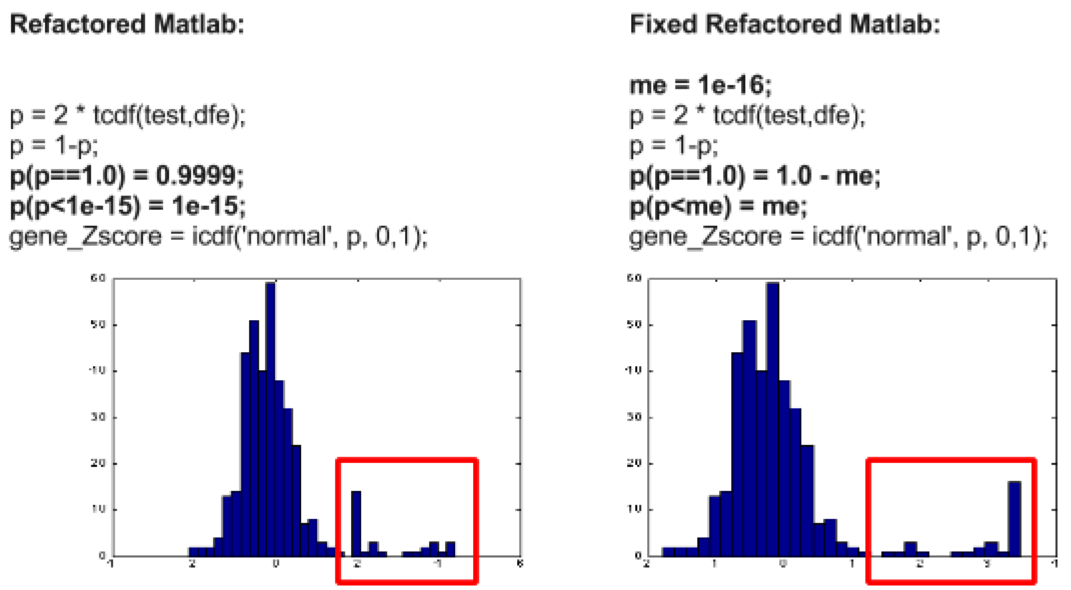


Figure 15: Idecker et al. z-score calculation as implemented in `geneScore`.

We use an optimized bubble sorting algorithm [Knuth1998]. We leverage code reuse from Numerical Recipes [Press1986]. In particular, we are concerned about numerical precision and stability. The level of visibility and rigor applied to Numerical Recipes elevate our probability of success. In particular we borrow two key elements: pseudo matrix inverse (`pinv`) and `tcdf` / `icdf`.

2.4 Z-score Bounds Corrections

Our unit tests had difficulty handling uncorrected bounds on p-values prior to icdf normal inversion in geneScore. In the refactoring process, we created genescore.m and moved the p-value inversion before the bounds corrections. The bounds corrections are needed when p-values are near 0 and 1 as the `icdf` of these values is `-inf` or `inf`. Figure 16 illustrates the unfair / asymmetrical correction to bounds, the region between 1 and 0.9999 is larger than between 1e-15 and 0. The refactored MATLAB shows a histogram of z-scores where a group of z-scores lie outside the standard normal distribution. The fix was to make symmetrical adjustments to the p values around 0 and 1. This moved the z-score spur to the right indicating some measurement in the gene expression data.



Refactored Matlab:

```
p = 2 * tcdf(test,dfe);
p = 1-p;
p(p==1.0) = 0.9999;
p(p<1e-15) = 1e-15;
gene_Zscore = icdf('normal', p, 0,1);
```

Fixed Refactored Matlab:

```
me = 1e-16;
p = 2 * tcdf(test,dfe);
p = 1-p;
p(p==1.0) = 1.0 - me;
p(p<me) = me;
gene_Zscore = icdf('normal', p, 0,1);
```

Figure 16: Correction to Z-score MATLAB calculation used as basis for C/C++ implementation.

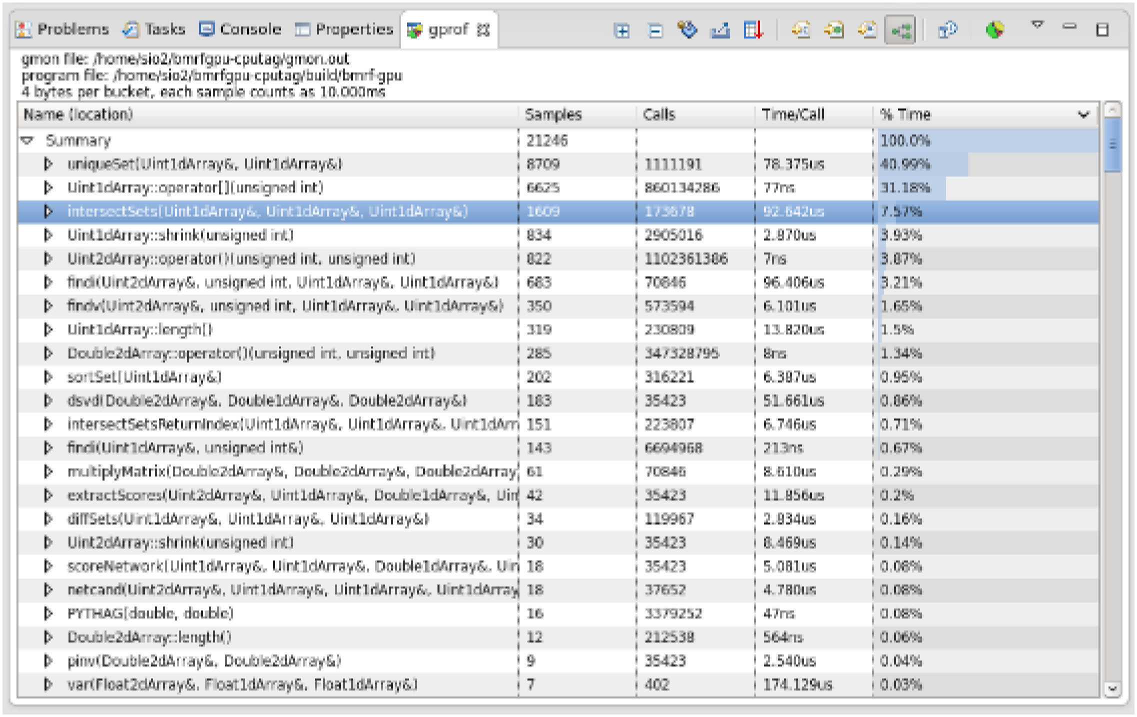
2.5 Implementation Optimizations

The [Loi2008] dataset is larger than the simulated dataset. The gene array is order 200 times larger and the PPi data is twice as large, Table 1 shows a comparison of the input variable dimensions. As many of our set functions like intersect, unique, and sort scale non-linearly due to algorithm complexity of $O(n^2)$ or worse. These effects will lead to less performance gain when the Loi dataset is used.

Variables	**Data Type**	**SIM**	**LOI**
geneArray	float	376x40	13799x190
geneIdArray	uint	376x1	13799x1
geneLabelArray	uint	1x40	1x190
ppiArray	uint	1825x2	2758x2
seedGeneIdArray	uint	1x1	193x1
zscore	double	376x1	13799x1

Table 1: Simulated vs. Loi dataset input dataset sizes. Format is WxH.

We then initiate the traditional profile, optimize, test, repeat loop for performance improvement. We used the GNU Profiler (`gprof`) which required a simple recompile with `-pg` flag [Fenlason1994]. Figure 17 illustrates the output from `gprof` as viewed from Eclipse. To summarize, `uniqueSet` and `intersectSets` represented 41%/77% (Simulated / Loi datasets) and 7.5%/16% respectively of BMRF execution time. Note the increase in total execution time by unique and intersect between the simulated and Loi datasets, 48% to 93% of execution time. This evidence suggests a confirmation of our hypothesis of O(n^2) or worse performance in some functions.

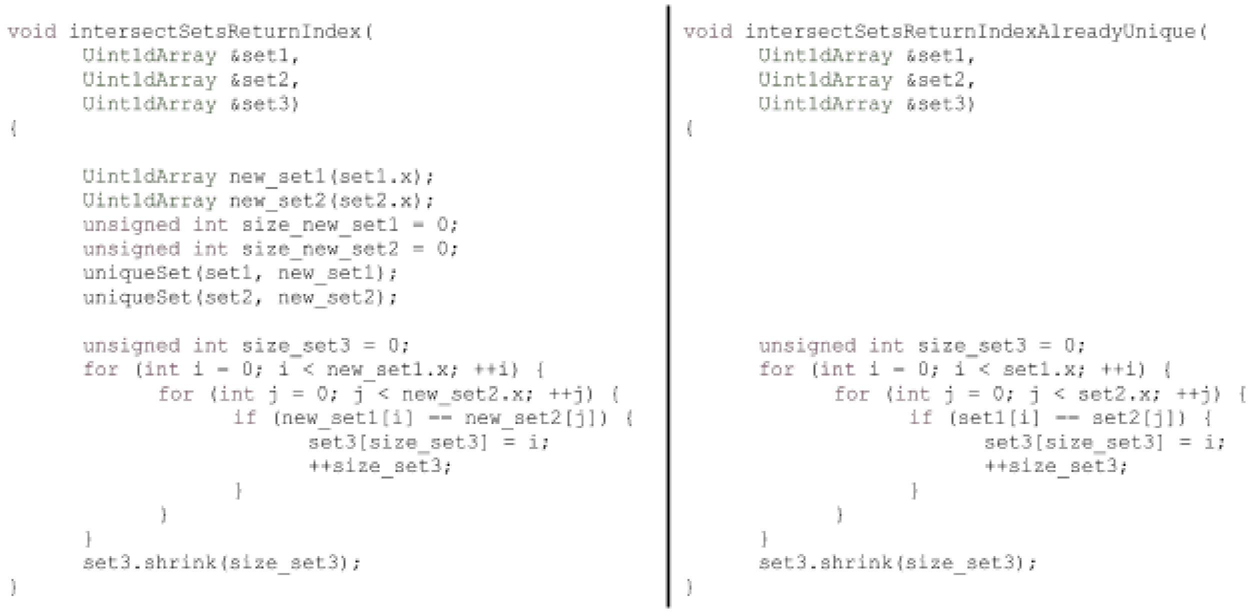


```
gmon file: /home/sio2/bmrfgpu-cputag/gmon.out
program file: /home/sio2/bmrfgpu-cputag/build/bmrf-gpu
4 bytes per bucket, each sample counts as 10.000ms
```

Name (location)	Samples	Calls	Time/Call	% Time
Summary	21246			100.0%
uniqueSet(Uint1dArray&, Uint1dArray&)	8709	1111191	78.375us	40.99%
Uint1dArray::operator[](unsigned int)	6625	860134286	77ns	31.18%
intersectSets(Uint1dArray&, Uint1dArray&, Uint1dArray&)	1609	173678	92.642us	7.57%
Uint1dArray::shrink(unsigned int)	834	2905016	2.870us	3.93%
Uint2dArray::operator()(unsigned int, unsigned int)	822	1102361386	7ns	3.87%
findi(Uint2dArray&, unsigned int, Uint1dArray&, Uint1dArray&)	683	70846	96.406us	3.21%
findv(Uint2dArray&, unsigned int, Uint1dArray&, Uint1dArray&)	350	573594	6.101us	1.65%
Uint1dArray::length()	319	230809	13.820us	1.5%
Double2dArray::operator()(unsigned int, unsigned int)	285	347328795	8ns	1.34%
sortSet(Uint1dArray&)	202	316221	6.387us	0.95%
dsvd(Double2dArray&, Double1dArray&, Double2dArray&)	183	35423	51.661us	0.86%
intersectSetsReturnIndex(Uint1dArray&, Uint1dArray&, Uint1dArr	151	223807	6.746us	0.71%
findi(Uint1dArray&, unsigned int&)	143	6694968	213ns	0.67%
multiplyMatrix(Double2dArray&, Double2dArray&, Double2dArray	61	70846	8.610us	0.29%
extractScores(Uint2dArray&, Uint1dArray&, Double1dArray&, Uin	42	35423	11.856us	0.2%
diffSets(Uint1dArray&, Uint1dArray&, Uint1dArray&)	34	119967	2.834us	0.16%
Uint2dArray::shrink(unsigned int)	30	35423	8.469us	0.14%
scoreNetwork(Uint1dArray&, Uint1dArray&, Double1dArray&, Uin	18	35423	5.081us	0.08%
netcand(Uint2dArray&, Uint1dArray&, Uint1dArray&, Uint1dArray	18	37652	4.780us	0.08%
PYTHAG(double, double)	16	3379252	47ns	0.08%
Double2dArray::length()	12	212538	564ns	0.06%
pinv(Double2dArray&, Double2dArray&)	9	35423	2.540us	0.04%
var(Float2dArray&, Float1dArray&, Float1dArray&)	7	402	174.129us	0.03%

Figure 17: Eclipse `gprof` view of simulated dataset. Far right column is the percentage of execution time spent on the function specified by row, sorted in descending order.

Following Amdahl's Law, we first worked to optimize `uniqueSet` being the largest consumer and thus giving us the highest potential for minimizing execution time. However, `uniqueSet` is never called directly but is called within `intersectSet` and intersectSetReturnIndex both of which are called frequently. The first critical observation is that the gene identifiers provided to mrfSearchNet are already sorted and are unique, as is the every result from netcand. This leads to the creation of intersectSetsReturnIndexAlreadyUnique which as the name implies skips the two uniqueSet calls on each of the two set arguments, Listing 2. This reduced function can replace two calls within mrfSearchNet.



```
void intersectSetsReturnIndex(
      Uint1dArray &set1,
      Uint1dArray &set2,
      Uint1dArray &set3)
{

      Uint1dArray new_set1(set1.x);
      Uint1dArray new_set2(set2.x);
      unsigned int size_new_set1 = 0;
      unsigned int size_new_set2 = 0;
      uniqueSet(set1, new_set1);
      uniqueSet(set2, new_set2);

      unsigned int size_set3 = 0;
      for (int i = 0; i < new_set1.x; ++i) {
            for (int j = 0; j < new_set2.x; ++j) {
                  if (new_set1[i] == new_set2[j]) {
                        set3[size_set3] = i;
                        ++size_set3;
                  }
            }
      }
      set3.shrink(size_set3);
}
```

```
void intersectSetsReturnIndexAlreadyUnique(
      Uint1dArray &set1,
      Uint1dArray &set2,
      Uint1dArray &set3)
{

      unsigned int size_set3 = 0;
      for (int i = 0; i < set1.x; ++i) {
            for (int j = 0; j < set2.x; ++j) {
                  if (set1[i] == set2[j]) {
                        set3[size_set3] = i;
                        ++size_set3;
                  }
            }
      }
      set3.shrink(size_set3);
}
```

Listing 2: intersectSetsReturnIndex with and without uniqueSet calls

Drilling down the stack, there are two very similar sequences of calls made involving intersections: 1) within netcand 2) within g_conn. They differ only in that (1) returns values and (2) returns indexes to values. The goal of the sequence is to build a set of genes that are connected with each gene in the previous candidate network. This requires looking at each column of the PPI array and selecting the corresponding connected protein / gene and selecting only those genes that are within the gene expression data. Our naive implementation did this with two finds, one union and an intersection (which called two uniques). Each find (`findv)` iterates through a column of the PPI array and returns the corresponding column if the value matches the search list. The output of each `findv` is then unioned which requires a sort and unique which can be optimized by assuming the data is sorted. Thus, we minimize the finds by looping once through the PPI and search list and return a merged list, we name this function `doublefindv.` Then we `sortSet` and `uniqueSetInplaceMustBeSorted` which runs at O(n) instead of O(n^2). This same approach then reused within `g_conn` but indices are returned instead of values. See Figure 18.

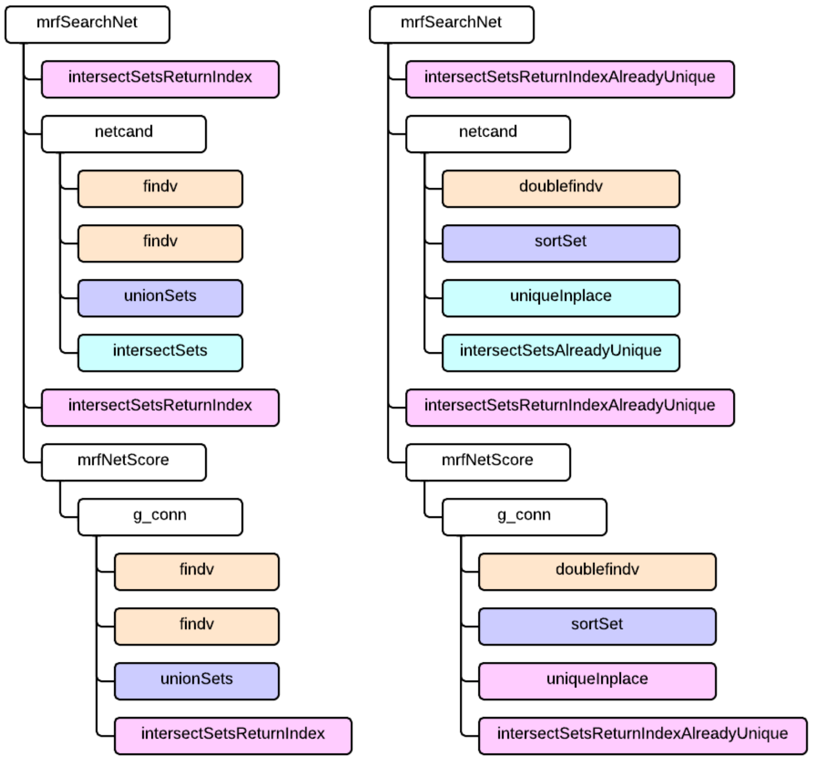


Figure 18: intersectSet usage in mrfSearchNet and dependencies. Left: before. Right: after unique reduction effort. Color indicates compatible functions on left and right.

Additionally, we replaced our naive bubble sort implementation in `sortSet` with `quicksort.` Bubble sort is O(n^2), where as qsort, a quick sort implementation found in

`<cstdlib.h>`, is O(nlog(n)) which offers a n/log(n) performance gain. Per the optimization cycle, we must examine the profiling results again.

Function Name	**% execution time**
intersectSets	20.14
dsvd	16.95
findi	16.62
intersectSetsAlreadyUnique	11.52
findi	9.72
intersectSetsReturnIndexAlreadyUnique	8.13
finddoublev	6.22
mrfnetscore	3.69
pinv	3.33
netcand	1.37
qsortCompare	0.86
diffSets	0.54
unionSets	0.41
uniqueSet	0.27
uniqueSetInplaceMustBeSorted	0.03

Table 2: profile data for simulated data after algorithm optimizations

If we compare Table 2 to Figure 19, the top consumers of execution time has changed. The `uniqueSet` function moved from 41% to 0.27%. `intersectSets` moved from 7.5% to 20%. This type of shift or "bubble up" can be common. As the overall execution time decreased, the share of execution time `intersectSets` holds increased. Our options for optimizing `intersectSets` however now resort to compiler optimizations and parallelism.

2.6 Compiler Optimizations

Modern compilers include multiple performance tuning flags, generally four ranging from no optimizations (level 0) to aggressive optimizations that can alter the results of some algorithms (level 3). Some compilers are open source while others are paid. Gcc is the former and Intel's 2013 Compiler Suite is the latter, however Intel provides a free license for academic staff and students.

Our C/C++ implementation uses basic objects and uses minimal additional code or wrappers unless absolutely necessary for readability. Regardless, it will benefit from compiler optimizations and most likely the compiler will be able to use SIMD extensions for the dense calculations in `mrfnetscore`. We have not made extensive use of inline function, our approach for optimization throughout this effort has been to find the largest performance increase for the lowest effort measured in time. Hand tuned C/C++ is a time intensive task and has diminishing returns as some optimizations can require specific processor architectures or features. Because of the large number of repeated `mrfSearchNet` calls we seek the performance increase as a result of exploiting concurrency.

2.7 Concurrency Optimizations

First step to realize concurrency optimizations is to identify independent tasks that can be executed at the same time. As our target processing element (PE) is any x86 processor or processors, so our unit of execution (UE) is relatively unencumbered in terms of number of registers and memory. Each bootstrap and baseline iteration can be run independently due to the nature of random resampling which produces independent data sets. The resampled data is shared, however, between all seed executions of `mrfSearchNet`. Conveniently, `mrfSearchNet` does not modify the data so each seed could share or we could duplicate the data without consequence to results. Furthermore, this implies a lack of race conditions and deadlocks and thus there is no need for synchronization except just before credibility analysis. Figure 19 illustrates moving the random resampling and gene score calculation before the baseline and bootstrap calls to `mrfSearchNet.`

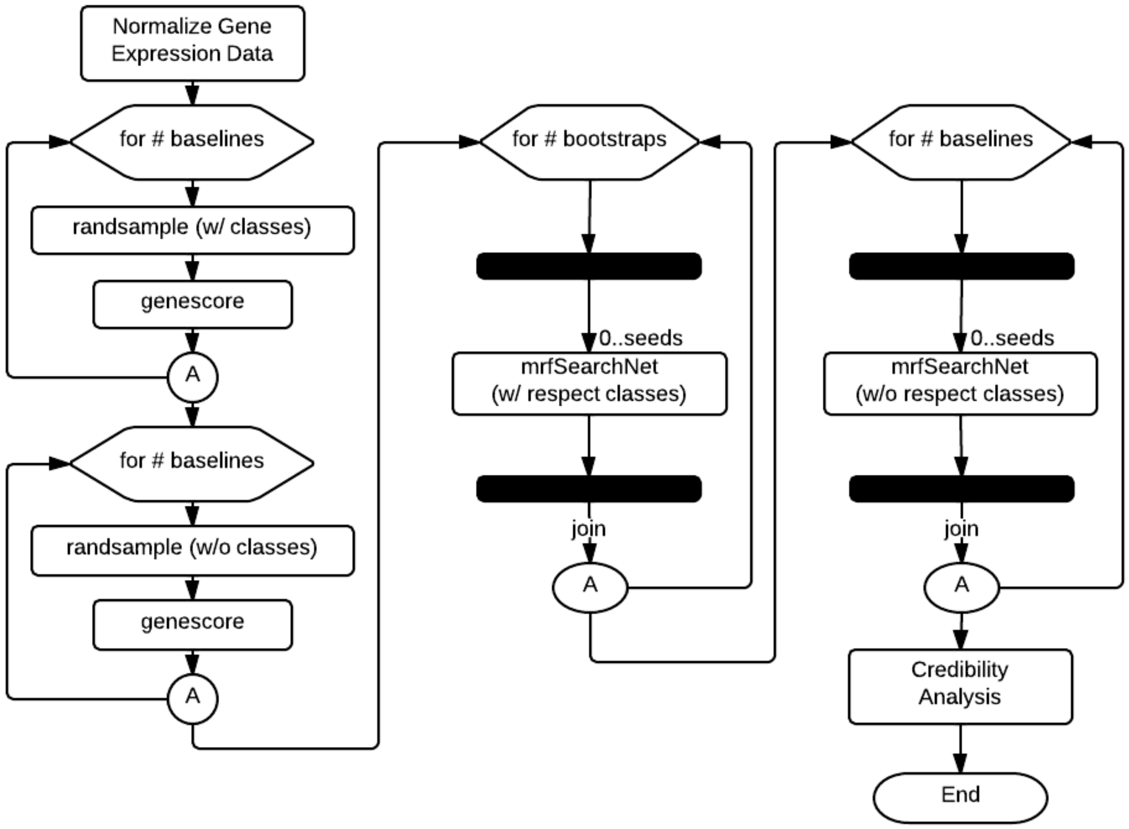


Figure 19: UML flow diagram of `BmrfHostAlgorithm::runHostMulti` showing one-dimensional parallelism by number of seeds.

We can express the single threaded runtime as:

$$T_{total}(1) = T_{setup} + T_{compute} + T_{final}$$

$$T_{total}(1) = T_{normal+zscores} + T_{bootstrap} + T_{baseline} + T_{ca}$$

We can estimate that normalization and z-score calculations and confidence analysis will take a very short period of time versus the two bootstrap runs. Thus our serial fraction is:

$$\gamma = \frac{T_{setup}+T_{final}}{T_{total}(1)} = \frac{T_{norma+zscores}+T_{ca}}{T_{total}(1)} =\approx 0$$

With an almost zero serial fraction, Amdahl's law can then be expressed as:

$$S(P) = \frac{1}{\gamma+\frac{1-\gamma}{P}} \Longrightarrow P$$

Which means that each additional processor added will result in a proportional speedup. However, given that SA terminates based on an annealing table driven by subnetwork score and given that `netcand` randomly produces a non-empty candidates, each call to `mrfSearchNet` will results in different execution time. This means that the P processors will not be fully utilized throughout each bootstrap iteration. A two-dimensional parallelism is required to maintain a 100% duty cycle on the processor where when all the last `mrfSearchNet` call within bootstrap 0 completes, the first `mrfSearchNet` call is started from bootstrap 1. We modify `BmrfHostAlgorithm::runHostMulti` to place all `mrfSearchNet` calls bootstrap x seed within a bootstrap or baseline section into a fifo thread pool, see Figure 20. We use boost threads with Philipp Henkel's `threadpool` add-on for boost (available on sourceforge.net).

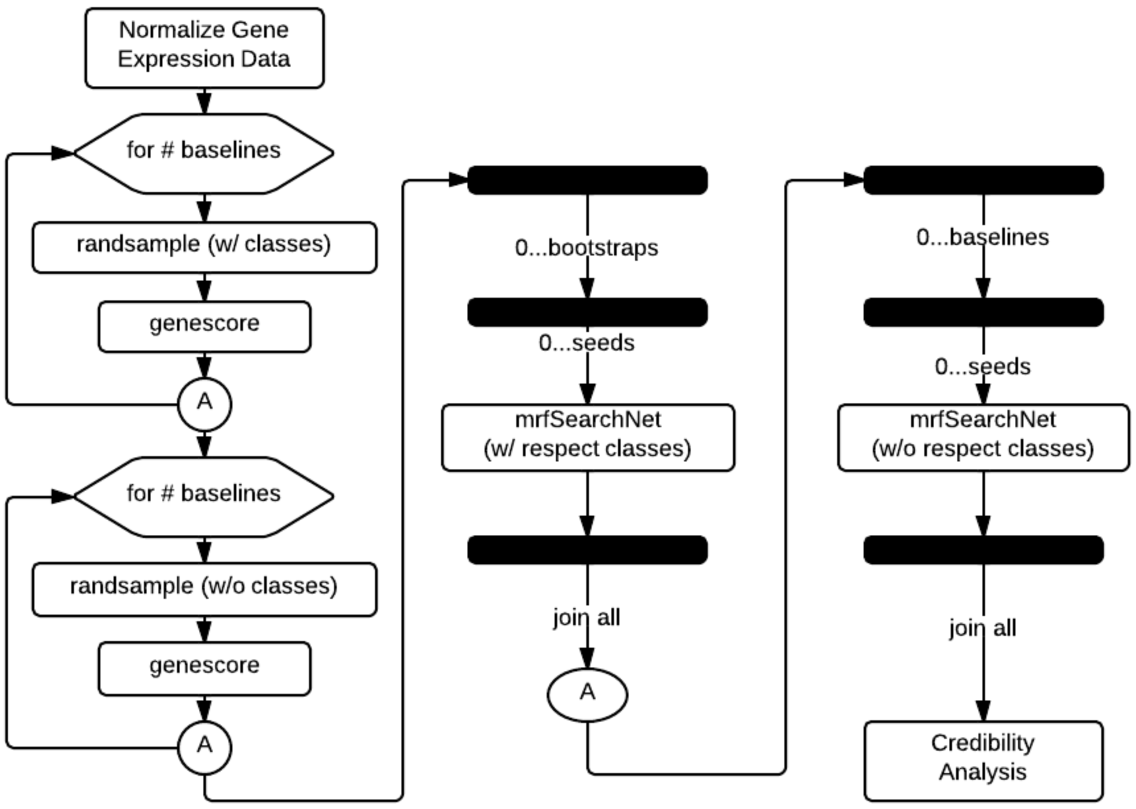


Figure 20: UML flow diagram of `BmrfHostAlgorithm::runHostMulti` showing two-dimensional parallelism by number of seeds and number of bootstraps or baselines.

2.8 GPU Concurrency Optimization Evaluation

Simulated annealing has been shown to provide speedups over CPU implementations in FPGA placement [Choong2010], in VLSI floorplanning [Han2011], in Gibbs sampling for finding Motifs [Yu2009]. A generic implementation, called CUISMANN, for the GPU is available as Open Source software (https://code.google.com/p/cusimann/). Gibbs sampling represents a good model for BMRF as it entails a Markov Chain Monte-Carlo procedure. However, because BMRF utilizes dependency information in each iteration of SA, the memory requirement for each thread is greater than that provided by the Fermi series of GPUs. We calculate the memory usage for `netcand` at 512KB for the Loi dataset whereas the Nvidia Fermi architecture has a configurable 16/48KB L1 cache/shared memory split [Nvidia2009]. This results in a global memory access pattern and thus a performance penalty erasing the gains over CPU. The Kepler architecture might provide a solution in that dynamic parallelism allows a GPU kernel to execute a GPU kernel. A notional design would be to implement a kernel to execute 2D-parallel seeds by bootstraps `mrfSearchNet` where each thread executes `netcand` as a kernel that divides the set operations across many threads reducing the L1 cache / shared memory usage.

Chapter 3

3. Experimental Results

3.1 Evaluating C/C++ for Correctness

Before we begin performance analysis, we first examined our C/C++ implementation against the MATLAB implementation as a reference or "gold standard". It is critical to ensure the stability of the algorithm because, as Breiman warns, bootstrapping with some classifiers can result in instabilities due to a reduced search space. In this case, if the annealing schedule is not correctly calibrated or if other problems exist in the SA implementation, the search space covered will be constrained leading to a different result between runs corresponding to different local optima. To evaluate this concern, we make ten runs of the C-code and compare the resulting subnetwork (and network scores) with each other and with the MATLAB output. Table 3 shows the output from the two implementations.

Impl.	**Results**
C/C++	2.638354 = [190 367 595 672 1387 1956 2099 2100 3065 3066 3265 5295 5469 5594 5595 5604 589 7157 8204 8841 9112 9612 9759 10014 10891 11331 27043 83933] 2.707399 = [190 595 672 1387 1956 2099 2100 3065 3066 3265 5295 5469 5594 5595 5604 5894 67 7157 8204 8841 9112 9612 9759 10014 10891 11331 27043 83933] 2.770982 = [190 595 672 1387 1956 2099 2100 3065 3066 3265 5295 5469 5594 5604 5894 6714 69 8204 8841 9112 9612 9759 10014 10891 11331 27043 83933] 2.797356 = [190 595 672 1387 1956 2099 2100 3065 3265 5295 5469 5594 5894 6714 6908 7157 82 9112 9612 9759 10014 10891 27043 83933]; 2.786953 = [190 595 672 1387 1956 2099 2100 3065 3265 5295 5469 5594 5595 5604 5894 6714 71 8841 9112 9612 9759 10014 10891 27043 83933]
MATLAB	2.731693 = [190 595 672 1387 1956 2099 2100 3065 3066 3265 5295 5469 5566 5594 5604 5894 67 7157 8204 8841 9112 9612 9759 10014 10891 27043 83933] 2.765700 = [190 595 672 1387 1956 2099 2100 3065 3265 5295 5469 5594 5604 5894 6714 6908 71 8841 9112 9612 9759 10014 10891 11331 27043 83933] 2.731693 = [190 595 672 1387 1956 2099 2100 3065 3066 3265 5295 5469 5566 5594 5604 5894 67 7157 8204 8841 9112 9612 9759 10014 10891 27043 83933] 2.696293 = [190 595 672 1387 1956 2099 2100 3065 3265 5295 5469 5594 5595 5604 5894 6714 6 7157 8204 8841 9112 9612 9759 10014 10891 11331 23013 27043 83933] 2.696293 = [190 595 672 1387 1956 2099 2100 3065 3265 5295 5469 5594 5595 5604 5894 6714 6 7157 8204 8841 9112 9612 9759 10014 10891 11331 23013 27043 83933]

Table 3: C/C++ vs. MATLAB implementations' output. Format: score = [gene ids]

We observe a result from C/C++ that remains stable over multiple executions and matches in both score and network identifiers with the MATLAB version. That is to say the subnetwork changes only by roughly 20% between executions in both implementations. With errors or implementation discrepancies, we have observed as much at 50% of the subnetwork change between executions. With a validated implementation, we have confidence in our implementation and can begin making optimizations.

3.2 Compiler Optimization

We ran our BMRF C/C++ single threaded implementation with implementation optimizations on both data sets varying the level of optimization 0 through 3 with both GCC and Intel's 2013 Compiler Suite. The results are shown in Table 4.

Compiler Optimization	**SIM (s)**	**LOI - 1 seed (s)**
Matlab	3490	10015
GCC -O0	444	5287
GCC -O1	99	1117
GCC -O2	74	643
GCC -O3	**69**	**643**
INTEL -O0	508	5999
INTEL -O1	90	930
INTEL -O2	85	899
INTEL -O3	81	874

Table 4: Comparison of GCC and Intel compiler optimization flags on 2.67 GHz Intel Core i7 920

We examined the resulting output against matlab truth We feel confident with the -O3 gcc optimization level achieving an optimization.

3.3 Simulated Dataset

We run the MATLAB and C/C++ implementations on a 2.67 GHz Intel Core i7 920, with no compiler optimizations and no parallelism. Our expectation is that the C/C++ implementation will run faster due to the performance losses associated with the inefficiencies of an interpreted language, as supported by columns two and three of Table 5.

SIM Dataset	MATLAB	Naive C/C++	Impl. Opt.	Comp. Opt
Normalize (s)	0.1	0.01	0.01	0
Bootstrap (s)	2090	572	291	41.8
Baseline (s)	1398	407	226	27.1
Credibility (s)	0.6	0.12	0.04	0
Total (~)	**58 min**	**16 min**	**9 min**	**69 sec**

Table 5: MATLAB vs. C-code execution times through non-concurrent optimizations on the simulated dataset on 2.67 GHz Intel Core i7 920.

We realize many fold improvements. We have summarized the speedup factor over MATLAB implementation across our optimizations in Table 6. We saw roughly a twofold decrease in execution time with our implementation optimizations. What is not clear is if a twofold decrease would be observed if the implementation and compiler optimizations were reversed. Regardless compiler optimizations provided a significant performance improvement.

Sim Dataset Speedups	Runtime (sec)	Speedup Factor
Matlab	3480	1.0
Naive C/C++	960	3.6
Impl. Opt.	540	6.4
Comp. Opt.	70	49.8

Table 6: C/C++ implementation performance vs. MATLAB on simulated dataset.

It should be mentioned that after each optimization step we thoroughly examined the output versus MATLAB's to ensure optimizations did not alter correctness. Armed with an optimized and validated version, we then apply a more difficult dataset.

3.4 Loi Dataset Evaluation

We knew from our compiler optimization analysis that the Loi dataset is more demanding than the simulated dataset. Table 7 illustrates the order 200 times larger dataset's effect on speedup factor.

Single Threaded	MATLAB	GCC -O3	Speedup Factor
SIM (s)	3490	70	49.8
Loi - 1 seed (s)	10015	645	15.5

Table 7: Speedups Matlab versus GCC on two datasets.

While diminished from 49x to 15x, the speedup on the Loi data remains respectable. At this point, we were confronted with a question of direction. Continue to optimize our C/C++ single threaded implementation to reduce big-O notation function complexity or attempt leverage concurrency identified in our design. We moved forward to our first multi-threaded implementation.

3.5 Evaluating 1D Concurrency

Parallelizing by 8 threads (Intel Core i7 920 has four cores each with HyperThreading for two threads per core) we would expect to observe 8x speedup; given Amdahl's law evaluated with the serial fraction approaching zero as demonstrated by section 2.7.

$$S(P) = \frac{1}{\gamma + \frac{1-\gamma}{P}} \Longrightarrow P$$

Again, using our Intel Core i7 920, we evaluated MATLAB versus C/C++ with 1 seed and with 10 seeds. The 1 seed leveraged no parallelism as our parallelism is by the number of seeds but the 10 seed performance run did exploit concurrency.

LOI - 1D Parallel	**Matlab**	**i7 - C/C++**	**Speedup**
1 seed (ST)	10015	645	15.5
10 seeds (MT)	37167	2431	15.3

Table 8: Execution performance of Loi dataset with 1 and 10 seeds with Multi-threaded (MT) implementation versus MATLAB.

We expected a speedup of 15.5*P where P is equal to 8. However our speedup over MATLAB remains at 15x. We can attribute this shortfall to two factors: 1) processor duty cycle 2) MATLAB optimizations.

Parallelizing by number seeds results in an inverse ramp processor utilization such that the duty cycle is 100% at the beginning of the for-#-seeds loop and then ramps down until all seeds have completed processing. This is a function of the parallel execution of simulated annealing with independently controlled annealing schedules driven by rand and subnetwork score. We empirically estimate 50% by studying the processor utilization graph available on our CentOS (Linux) desktop. A notional duty cycle is shown in Figure 21.

To say MATLAB is unoptimized is incorrect. It performs sub-optimally compared to C/C++ implementation but Mathworks has spent considerable effort to improve its performance. MATLAB has a variable that controls the number of threads used in BLAS and JIT sub-components [Loren2007]. Our particular instance was configured with a value of 4. The effect of which can be seen in the second row of Table 8. If we ran the MATLAB implementation of for one seed ten times, we would have observed a factor of 10 increase in execution time. However our experiment showed only a 3.7 factor of increase in execution.

With MATLAB performing 3.7x better than expected and our duty cycle being estimated at half, we can approximately account for the 8x shortfall. We decided to gather an additional set of data points. Using a 64-core AMD Opteron 6276 machine, we examined re-examined our C/C++ implementation's performance. We had one minor technical challenge to arrive at the best performance with 64 processors. OpenMP has an implicit limit which only allowed, at most, 50 processors to be utilized. Increasing the number of threads to 193 allowed us to utilizes all 64 processors. Note, we also observed a processor usage duty cycle less than 100%. A notional duty cycle is shown in Figure 21.

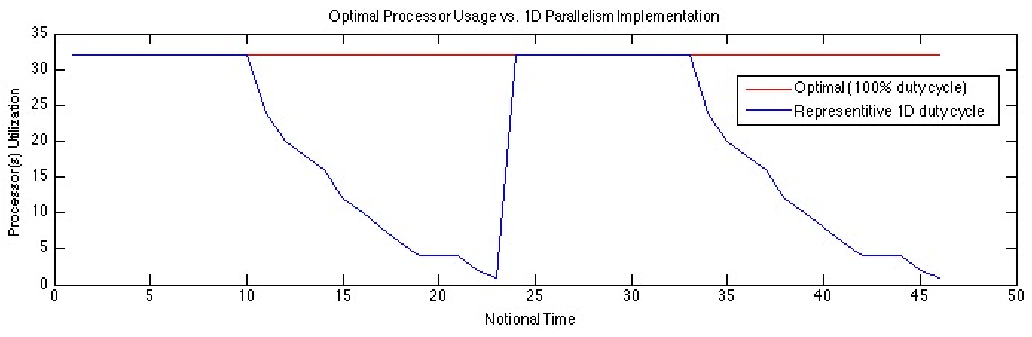


Figure 21: A notional graph of the duty cycle observed with 1D parallelism implementation.

Looking at Table 9, we see the performance from our MATLAB versus i7 repeated in the first two columns and the addition of the Opteron performance in the third column. We were able to also run the full Loi dataset (all 193 seed genes on the i7 and Opteron) thus the addition of the last row.

LOI - 1D Parallel	**MATLAB**	**i7 - C/C++**	**Opteron - C/C++**
1 seed (ST)	10015	645 (15.5x)	955 (10.5x)
10 seeds	37167	2431 (15.3x)	2173 (17.1x)
193 seeds	522404 (est.)	36792 (14.2x)	6891 (75.8x)

Table 9: one-dimension concurrency exploitation on i7 and 64-core Opteron versus MATLAB.

For the single threaded or 1 seed experiments, we can identify that the i7 has a 5x performance gain over the Opteron processor. However, moving to the 10 seeds experiments, the Opteron overcomes this deficit in single threaded performance by utilizing two extra cores which implies that our implementation possesses some contention between HyperThreads on the Intel Core i7. With ten times more work (10 seeds vs. 1 seed), the Opteron was able to only deliver a 7.1x speedup delta which helps to confirm our hypothesis that the processors are being underutilized.

We also present an estimated MATLAB performance on the 193 seeds on the 97 at 145 hours. Our calculation is driven by the single threaded performance (10015 seconds) multiplied by the increased units of work (193) divided by the calculated MATLAB performance increase due to the number of threads variable (3.7). Using this estimate, we observe a 75.8 fold improvement using the 64-core Opteron. This presents a drastic improvement and an asset for the CBIL team but not yet an optimal parallelized version.

3.6 Evaluating 2D Concurrency

Building on the performance improvements achieved by one-dimensional parallelism, our goal is to increase processor duty cycle to optimum. We will evaluate our threadpool implementation which added parallelism across bootstraps in addition to seeds. Table 10 presents the same MATLAB performance results but against this two-dimensional concurrency exploiting implementation.

LOI - 2D Parallel	MATLAB	i7 - C/C++	Opteron - C/C++
1 seed	10015	211 (47.5x)	57 (175.7x)
10 seeds	37167	1698 (21.8x)	260 (142.9x)
193 seeds	522404 (est.)	34319 (27.1x)	4404 (118.6x)

Table 10: 2D parallelism performance against MATLAB, using i7 (P=8) and Opteron (P=64) machines.

The single seed performance numbers are impressive, however, we do realize the comparison is a single threaded version versus a heavily multi-threaded versions. We particularly want to notice performance gains over the one-dimensional implementation. Table 11 illustrates this. Note the far right column is the difference between the speedup factors of 2D and 1D implementations. Recall that we identified a less than 100% duty cycle and estimated it at approximately half. The far right column shows us a relative speedup factor of 1.4. Expressed differently, the two-dimensional implementation utilized 40% more of the processor. Indicating our estimate of 50% duty cycle was missed actual by 10%.

Speedups (i7)	**C/C++ 1D**	**C/C++ 2D**	**Relative Speedup**
1 seed	15.5	47.5	3.1
10 seeds	15.3	21.8	1.4
193 seeds	14.2	27.1	1.9

Table 11: Speedups of C/C++ 1D and 2D implementations on i7. Relative Speedup indicates gains in efficient usage of processors.

Moving to the Opteron, we observe a similar phenomenon in Table 12. The 193 seed experiment shows a 60% improvement in processor utilization while the 10 seed boasts at 740% increase. The one-dimensional implementation when run on the Opteron was limited to 10 processing cores so the ability to use all 64 processing cores presents a significant performance improvement.

Speedups (Opteron)	**C/C++ 1D**	**C/C++ 2D**	**Relative Speedup**
1 seed	955	57	16.8
10 seeds	2173	260	8.4
193 seeds	6891	4404	1.6

Table 12: Speedups of C/C++ 1D and 2D implementations on Opteron. Relative Speedup indicates gains in efficient usage of processors.

Chapter 4

4. Conclusions

4.1 Summary

We refactored the MATLAB reference implementation to prepare for a successful optimization effort. Then decomposed the algorithm to basic components, arriving at a design for a C/C++ implementation. Using unit testing, we implemented a naive C/C++ application. We then tested against our C/C++ implementation, using MATLAB as a "gold standard", and confirmed validity and implementation stability which was a significant concern. With a valid C/C++ implementation, we began a series of optimizations: implementation, compiler, and concurrency (one-dimensional and two-dimensional). We demonstrated many-fold performance improvements at each step, using two computers to evaluate our performance: a desktop machine and a many-core server. Taking time to carefully examine the performance versus the predicted performance improvement calculated using Amdahl's law. We discovered multiple discrepancies in our predictions and used both machines to exploit and evaluate the hypothesis and further encourage optimizations in our implementation.

We summarize our performance improvements in Figure 22. Our main goal was to improve the workflow of CBIL's research team. Our optimized implementation resulted in the ability to run BMRF on a desktop machine overnight rather than multiple days. Additionally, using the large Opteron server, a researcher can perform what previously took almost a full day to a lunch break.

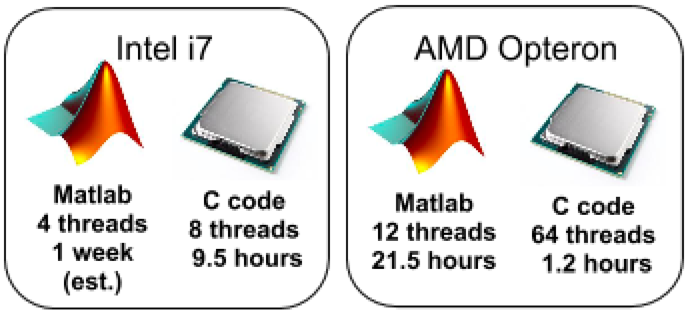


Figure22: Performance summary of best implementations

4.2 Future Work

An optimized C/C++ implementation and concurrency analysis enables 1) commodity hardware 2) larger data sets 3) optimized cluster performance 4) coprocessors.

Commodity hardware such as personal computers are being equipped with and increasingly number of processing cores capable of parallelism approaching the performance of large high performance computers of yesteryear. The implementation presented here exploits parallelism effectively, enabling year over year improvement with each technology refresh at the desktop computer price point. Researchers can also increase the size, dimensionality, fidelity or complexity of their algorithmic pursuits in integrative network analysis.

The implementation presented here also opens the door for larger level of parallelism with multiple computers in a cluster of cloud configuration. This is often considered a MIMD architecture. A simple example would be to execute the bootstrap and baseline sections of BMRF on separate machines, resulting in a 2x speedup minus network transfer cost.

Message passing interface (MPI) could be used to distribute number of bootstrap by number of seeds to many computers resulting in a many fold speedup dependent on the number of network transfers, homogeneity of cluster nodes, and size of tasks distributed. This last approach is particularly suited well should the data set be expanded.

Lastly, the advent of advance of highly parallel coprocessors remains an attractive option to further expand the capability of desktop machines. The barrier to utilize Nvidia Tesla (GPUs) and Intel Phi coprocessors is dramatically reduced with an optimized and parallelized C/C++ implementation.

There are further optimizations that can improve performance. Li Chen mentions, as we do as well, the ability to terminate the SA early based on when last update was made. This optimization could reduce the execution time by 30%. The profiling results indicate that a few functions could be further optimized and libraries such as BLAS or Intel's MKL could improve matrix / vector operations as their implementations are highly optimized for specific x86 architectures.

References

[Akay2007] M. Akay, *Genomics and Proteomics Engineering in Medicine and Biology*. Piscataway, NJ: IEEE Press 2007.

[Allison2006] D. B. Allison, X. Cui, G. P. Page, M. Sabripour, *Microarray data analysis: from disarray to consolidation and consensus*. Nature Reviews Genetics 7, 55-65, January 2006.

[Amdahl1967] G. M. Amdahl, *Validity of the single processor approach to achieving large scale computing capabilities*. AFIPS spring joint computer conference, 1967.

[Berger1998] B. Berger, T. Leighton, *Protein Folding in Hydrophobic-Hydrophilic (HP) Model is NP-Complete. Journal of Computational Biology*, Vol. 5, Num. 1, 1998, Mary Ann Liebert, Inc., Pp 27-40.

[Breiman1996] L. Breiman, (1996) *Bagging Predictors.* Machine Learning 24: 123–140.

[Buck2004] I. Buck, T. Foley, D. Horn, J. Sugerman, K, Fatahalian, M.Houston, P. Hanrahan, *Brook for GPUs: stream computing on graphics hardware*. ACM Trans. Graph., Vol. 23, Num. 3, Pp. 777-786, 2004.

[Buris2011] Biological and Environmental Research Information System (BURIS),
About the Human Genome Project Available at:
http://www.ornl.gov/sci/techresources/Human_Genome/project/about.shtml, Modified 2011 AUG 22, Accessed 2013 APR 19.

[Chen2012] L. Chen, J. Xuan, R. Riggins, Y. Wang, R. Clarke, *Identifying protein interaction subnetworks by a bagging Markov random field-based method*, Nucleic Acids Research, September 2012.

[Choong2010] A. Choong, R. Beidas, J. Zhu, *Parallelizing Simulated Annealing-Based Placement using GPGPU*. IEEE Int'l Conference on Field Programmable Logic and Applications, 2010.

[Chuang2007] H. Y. Chuang, E. Lee, Y. T. Liu, D. Lee, T. Ideker, *Network-based classification of breast cancer metastasis*. Mol Syst Biol, 3, 140, 2007.

[Crick1970] F. Crick, *Central Dogma of Molecular Biology*. Nature, Vol 227, Aug 8 1970.

[Dittrich2008] M. T. Dittrich, G. W. Klau, A. Rosenwald, T. Dandekar, T. Muller, *Identifying functional modules in protein-protein interaction networks: an integrated exact approach*. Bioinformatics, Vol. 24, Issue 13, i223-231, 2008.

[Dove2006] K. Dove, *Nvidia Unveils CUDA^{TM}-The GPU Computing Revolution Begins*. Online Press Release, November 8 2006.

[Duda2001] R. O. Duda, P. E. Hart, D. G. Stork, *Pattern Classification, Second Edition*. New York, NY: John Wiley & Sons, 2001.

[Fenlason1994] J. Fenlason, R. Stallman, *GNU Profiler.* Available at:
http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html, Accessed 2013 ARP 19.

[Flynn1972] M. J. Flynn, *Some Computer Organizations and Their Effectiveness*. IEEE Transactions on Computers, Vol C-21, No. 9, September 1972.

[Han2011] Y. Han, S. Roy, K. Chakraborty, *Optimizing Simulated Annealing on GPU: A Case Study with IC Floorplanning*. IEEE 12th Int'l Symposium on Quality Electronic Design, 2011.

[Hanash2004] S. Hanash, *Integrated global profiling of cancer*, Nature Reviews Cancer, Vol. 4, Pp. 638-44, Aug 2004.

[Hoheisel2006] J. D. Hoheisel, *Microarray technology: beyond transcript profiling and genotype analysis*. Nature Reviews Genetics 7, 200-210, March 2006.

[Ideker2002] T. Ideker, O. Ozier, B. Schwikowski, A. F. Siegel, *Discovering regulatory and signalling circuits in molecular interaction networks*. Bioinformatics, 18 Suppl 1, S233-240, 2002.

[Intel2011] Intel, Processors: Define SSE2, SSE3, SSE4.
http://www.intel.com/support/processors/sb/CS-030123.htm. Accessible on: 2013 APR 19.

[Khurana2003] T. S. Khurana, K. E. Davies, *Pharmacological strategies for muscular dystrophy*. Nat Rev Drug Discov, Vol 2, Issue 5, 379-390, May 2003.

[Kim2003] N.S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, V. Narayana, *Leakage current: Moore's law meets static power*. IEEE Computer, Vol. 36, Issue 12, Pp. 68-72, December 2003.

[Kirkpatrick1983] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi, *Optimization by Simulated Annealing*. Science, New Series, Vol. 220, No. 4598. (May 13, 1983), pp. 671-680.

[Knuth1998] D. E. Knuth, *The Art of Computer Programming Volume 3 / Sorting and Searching Second Edition*. Boston, MA,: Addison-Wesley 1938.

[Loi2008] S. Loi, B. Haibe-Kains, C. Desmedt, P. Wirapati, F. Lallemand, A. M. Tutt, C. Gillet, P. Ellis, K. Ryder, J. F. Reid, M. G. Daidone, M. A. Pierotti, E. M. Berns, M. P. Jansen, J. A. Foekens, M. Delorenzi, G. Bontempi, M. J. Piccart, and C. Sotiriou, *Predicting prognosis using molecular profiling in estrogen receptor-positive breast cancer treated with tamoxifen*, BMC Genomics, vol. 9, p. 239, 2008.

[Loren2007] S. Loren, *Loren on the Art of MATLAB: Controlling Multithreading,* http://blogs.mathworks.com/loren/2007/09/12/controlling-multithreading/#1, Accessed: 2013 APR 19.

[Mattson2005] T. G. Mattson, B. A. Sanders, B. L. Massingill, Patterns for Parallel Programming. Boston, MA: Pearson Education, 2005.

[Mering2003] C. von Mering, M. Huynen, D. Jaeggi, S. Schmidt, P. Bork, and B. Snel, "STRING: a database of predicted functional associations between proteins," Nucleic Acids Res, vol. 31, pp. 258-61, Jan 1 2003.

[Mishra2006] G. R. Mishra, M. Suresh, K. Kumaran, N. Kannabiran, S. Suresh, P. Bala, K. Shivakumar, N. Anuradha, R. Reddy, T. M. Raghavan, S. Menon, G. Hanumanthu, M. Gupta, S. Upendran, S. Gupta, M. Mahesh, B. Jacob, P. Mathew, P. Chatterjee, K. S. Arun, S. Sharma, K. N. Chandrika, N. Deshpande, K. Palvankar, R. Raghavnath, R. Krishnakanth, H. Karathia, B. Rekha, R. Nayak, G. Vishnupriya, H. G. Kumar, M. Nagini, G. S. Kumar, R. Jose, P. Deepthi, S. S. Mohan, T. K. Gandhi, H. C. Harsha, K. S. Deshpande, M. Sarker, T. S. Prasad, and A. Pandey, "Human protein reference database--2006 update," Nucleic Acids Res, vol. 34, pp. D411-4, Jan 1 2006.

[Moore1965] G. E. Moore, *Cramming more components onto integrated circuits. Electronics Magazine*, Vol. 38, Num. 8, April 19 1965.

[Nvidia2009] Nvidia, *Whitepaper Nvidia's Next Generation CUDA™ Compute Architecture: Fermi*. Nvidia Corporation, 2009.

[Nvidia2012] Nvida, *Whitepaper NVIDIA's Next Generation CUDA™ Compute Architecture: Kepler™ GK110*.

[OpenMP2008] OpenMP Architecture Review Board, OpenMP Application Program Interface, Version 3.0. May 2008.

[Press1986] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, *Numerical Recipes, The Art of Scientific Computing, Third Edition*. New York, NY:Cambridge University Press, 1986.

[Pugh2005] Ken Pugh, *Prefactoring*. Sebastopol, CA:O'Reilly Media, 2005.

[Simoes2012] R. de Matos Simoes, F. Emmert-Streib, (2012) *Bagging Statistical Network Inference from Large-Scale Gene Expression Data*. PLoS ONE 7(3): e33624.

[Taylor2009] I. W. Taylor, R. Linding, D. Warde-Farley, Y. Liu, C. Pesquita, D Faria, S. Bull, T. Pawson, Q. Morris, J. L. Wrana, *Dynamic modularity in protein interaction networks predicts breast cancer outcome*. Nature Biotechnology 27, 199-204, 2009.

[Wetterstrand2013] K. Wetterstrand, DNA Sequencing Costs: Data from the NHGRI Genome Sequencing Program (GSP) Available at: www.genome.gov/sequencingcosts. Accessed 2013 APR 19.

[Yakhini2011] Z. Yakhini, I. Jurisica, *Cancer computational biology*. BMC Bioinformatics 2011 12:120.

[Yu2009] L. Yu, Y. Xu, *A Parallel Gibbs Sampling Algorithm for Motif Finding on GPU*. IEEE Int. Symposium on Parallel and Distributed Processing with Applications, 2009.

Appendix A

The repository containing the C/C++ source code for this effort is approximately 5,000 Source Lines of Code (SLOC) {according to David A. Wheeler's SLOCCount application} and is available by request <otto.barnes@vt.edu>.

Due to copyright restrictions, portions of the software will be removed: Numerical Methods pinv, icdf, tcdf; Loi dataset.

Repository Layout:

```
/doc - documentation
/matlab - refactored matlab
        sim - simulated dataset
/src
        libbmrfc - library of bmrf c code
        libbmrfgpu - library of bmrf GPU code (uses libbmrfc)
        bmrfc-tests - unit tests for libbmrfc
        bmrfcpu - cpu application for running bmrf (uses libbmrfc)
        bmrfgpu - gpu application for running bmrf (uses libbmrfgpu)
        CMakeLists.txt - cmake build file
/Makefile - quick access to common build functions (calls cmake)
```