Masthead Logo

Animal Science Publications Animal Science

2016

# High-performance epistasis detection in quantitative trait GWAS

Nathan T. Weeks
*Iowa State University*, weeks@iastate.edu

Glenn R. Luecke
*Iowa State University*, grl@iastate.edu

Brandon M. Groth
*Iowa State University*, bmgroth@iastate.edu

Marina Kraeva
*Iowa State University*, kraeva@iastate.edu

Li Ma
*Iowa State University*

***See next page for additional authors***

# High-performance epistasis detection in quantitative trait GWAS

**Abstract**

epiSNP is a program for identifying pairwise single nucleotide polymorphism (SNP) interactions (epistasis) in quantitative-trait genome-wide association studies (GWAS). A parallel MPI version (EPISNPmpi) was created in 2008 to address this computationally expensive analysis on large data sets with many quantitative traits and SNP markers. However, the falling cost of genotyping has led to an explosion of large-scale GWAS data sets that challenge EPISNPmpi's ability to compute results in a reasonable amount of time. Therefore, we optimized epiSNP for modern multi-core and highly parallel many-core processors to efficiently handle these large data sets. This paper describes the serial optimizations, dynamic load balancing using MPI-3 RMA operations, and shared-memory parallelization with OpenMP to further enhance load balancing and allow execution on the Intel Xeon Phi coprocessor (MIC). For a large GWAS data set, our optimizations provided a 38.43× speedup over EPISNPmpi on 126 nodes using 2 MICs on TACC's Stampede Supercomputer. We also describe a Coarray Fortran (CAF) version that demonstrates the suitability of PGAS languages for problems with this computational pattern. We show that the Coarray version performs competitively with the MPI version on the NERSC Edison Cray XC30 supercomputer. Finally, the performance benefits of hyper-threading for this application on Edison (average 1.35× speedup) are demonstrated.

**Keywords**

Xeon Phi coprocessor, epistasis, Coarray Fortran, MPI, OpenMP

**Comments**

This is a manusript of an article published as Weeks, Nathan T, Glenn R Luecke, Brandon M Groth, Marina Kraeva, Li Ma, Luke M Kramer, James E Koltes, and James M Reecy. "High-Performance Epistasis Detection in Quantitative Trait GWAS." *The International Journal of High Performance Computing Applications* (2016). doi:10.1177/1094342016658110.

**Authors**

Nathan T. Weeks, Glenn R. Luecke, Brandon M. Groth, Marina Kraeva, Li Ma, Luke M. Kramer, James E. Koltes, and James M. Reecy

Journal Title
XX(X):1–13
©The Author(s) 2015
Reprints and permission:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

# High Performance Epistasis Detection in Quantitative Trait GWAS

Nathan T. Weeks[1], Glenn R. Luecke[1], Brandon M. Groth[1], Marina Kraeva[2], Li Ma[4], Luke M. Kramer[3], James E. Koltes[5], James M. Reecy[3]

**Abstract**

epiSNP is a program for identifying pairwise single nucleotide polymorphism (SNP) interactions (epistasis) in quantitative-trait genome-wide association studies (GWAS). A parallel MPI version (EPISNPmpi) was created in 2008 to address this computationally-expensive analysis on large data sets with many quantitative traits and SNP markers. However, the falling cost of genotyping has led to an explosion of large-scale GWAS data sets that challenge EPISNPmpi's ability to compute results in a reasonable amount of time. Therefore, we optimized epiSNP for modern multi-core and highly-parallel many-core processors to efficiently handle these large data sets. This paper describes the serial optimizations, dynamic load balancing using MPI-3 RMA operations, and shared-memory parallelization with OpenMP to further enhance load balancing and allow execution on the Intel Xeon Phi coprocessor (MIC). For a large GWAS data set, our optimizations provided a 38.43X speedup over EPISNPmpi on 126 nodes using 2 MICs on TACC's Stampede Supercomputer. We also describe a Coarray Fortran (CAF) version that demonstrates the suitability of PGAS languages for problems with this computational pattern. We show that the Coarray version performs competitively with the MPI version on the NERSC Edison Cray XC30 supercomputer. Finally, the performance benefits of Hyper-Threading for this application on Edison (average 1.35X speedup) are demonstrated.

**Keywords**

Xeon Phi coprocessor, epistasis, Coarray Fortran, MPI, OpenMP

## 1 Introduction

A genome-wide association study (GWAS) statistically associates genetic variations with phenotypes (observable traits) in a group of individuals. These genetic variations, called single nucleotide polymorphisms (SNPs), can be detected at relatively low cost using SNP genotyping arrays. Sometimes, different alleles at a single location in the genome are associated with qualitative traits (traits that can be discretely categorized, such as hair color, blood type, or normal vs. diseased) or quantitative (continuous) traits (e.g., height, weight). Other times, it is the interaction of specific alleles in two (or more) locations in the genome that determines a given trait; this phenomenon is known as epistasis.

epiSNP (Ma et al. 2008) is a program for detecting epistasis between pairs of SNP alleles in quantitative-trait GWAS.* Epistasis detection can be computationally demanding: pairwise SNP marker tests used to identify epistasis require $\mathcal{O}(N^2)$ pairwise genetic marker tests to be performed for each trait. In addition, a large sample size (i.e., a large number of individuals) is required to have sufficient statistical power to identify real epistasis. Without parallel computing, only data sets of a modest size can be processed in a reasonable time frame. In 2008, a distributed-memory parallel version of epiSNP (EPISNPmpi) was created using the Message Passing Interface (MPI) to detect epistasis in what was considered large-scale GWAS at the time.

However, the decreasing cost of SNP genotyping has caused an explosion in the number of SNPs used in GWAS, taxing the ability of EPISNPmpi to computationally identify epistasis. To address the performance challenges posed by increasingly-larger GWAS data sets, the authors recently augmented the original MPI program with OpenMP-based shared-memory parallelism to reduce memory footprint, enhance load balancing, and allow execution on the Intel Xeon Phi (Luecke et al. 2015). This hybrid MPI+OpenMP code, which featured extensive serial optimizations, used static load balancing of MPI processes, and dynamic OpenMP scheduling within each MPI process. This resulted in much better load balancing and performance than the original pure-MPI EPISNPmpi. This program was successfully run on the Stampede supercomputer at the Texas Advanced Computing Center using both host CPUs and two Intel Xeon Phi coprocessors per node, resulting in a 36X speedup over the original pure MPI code using 126 nodes. However, it still exhibited some load imbalance when running solely on host CPUs, and required the user to

[1] Department of Mathematics, Iowa State University, USA
[2] IT Services Academic Technologies, Iowa State University, USA
[3] Department of Animal and Avian Sciences, University of Maryland, USA
[4] Department of Animal Science, Iowa State University, USA
[5] Department of Animal Science, University of Arkansas, USA

**Corresponding author:**
Nathan T. Weeks, Department of Mathematics, Iowa State University, Ames, IA, 50011, USA.
Email: weeks@iastate.edu

*Information on how to obtain epiSNP is available at http://animalgene.umn.edu/episnp/

estimate the performance of the Xeon Phi relative to the host CPUs in order to use the coprocessors effectively.

This paper expands upon our previous work (Luecke et al. 2015) by introducing dynamic load balancing. Two different approaches were evaluated: one using Coarray Fortran (CAF), a parallel programming feature added to the Fortran 2008 language standard; and the other MPI-3 one-sided communication. Our updates resulted in a single codebase from which one of several parallel versions can be created via compile-time preprocessor directives. Compiled as a serial or OpenMP executable, our optimized epiSNP can handle modest GWAS data sets on a multi-core workstation, while hybrid MPI3+OpenMP and CAF+OpenMP versions scale up to solve much larger data sets on a compute cluster.

### 1.1 Paper Organization

The rest of the paper is organized as follows. Section 2 discusses the differences between the work described in this paper and related work. Section 3 describes the result of performance profiling to identified bottlenecks in EPISNPmpi. Execution on the Intel Xeon Phi is discussed in Section 4. Section 5 details many of the serial optimizations that were employed to achieve a 15X serial speedup. Section 6 describes OpenMP-based multi-threading, which improved load balancing within a node, reduced memory usage, and made execution on accelerators feasible. Section 7 covers distributed-memory parallelism using MPI and CAF to provide dynamic load balancing. Section 8 measures the impact of the optimizations on application performance. Finally, the results are summarized in section 9.

## 2 Related Work

While epistasis detection in quantitative trait GWAS has been explored on single, standalone GPUs (Pütz et al. 2013), the authors are not aware of any other published software that utilizes Xeon Phi coprocessors. Subsequent to EPISNPmpi, the authors are aware of only one effort to detect epistatic interactions in quantitative-trait GWAS utilizing distributed-memory parallel computing (Koesterke et al. 2011), though the software itself was never made publicly available. This software statically assigns all pairwise iterations for one or more traits to each MPI process, and so unlike epiSNP, requires multiple traits to use distributed-memory parallelism.

A problem related to quantitative-trait GWAS (where the traits are continuous) is case-control GWAS (where the traits can be represented with binary values; e.g., normal vs. diseased). For case-control GWAS, epistasis detection has been performed on compute clusters using a number of parallel programming paradigms, including Hadoop MapReduce (Zhou et al. 2013) and MPI+OpenMP (Goudey et al. 2015), both of which utilized a static mapping of SNP pairs to reducers (Hadoop) or processes (MPI). The recent PGAS language UPC++ was used to implement dynamic load balancing between up to 24 GPUs (González-Domínguez et al. 2015), 2 GPUs and a Xeon Phi (Gonzalez-Dominguez et al. 2015), and 128 FPGAs (González-Domínguez et al. 2015); however, the host processors were used only for coordination, and not concurrent computation with the accelerators.

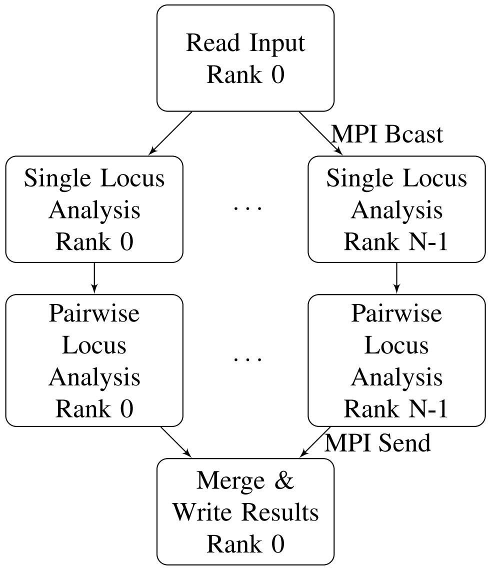


**Figure 1.** EPISNPmpi workflow. All processes are single-threaded. Code profiling revealed that over 99% of the run time was spent in the pairwise locus analysis.

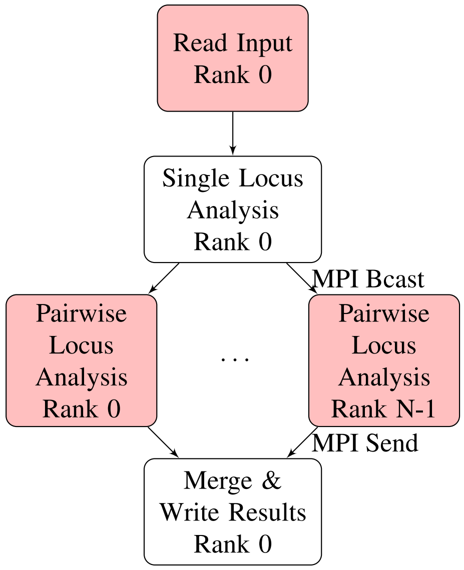
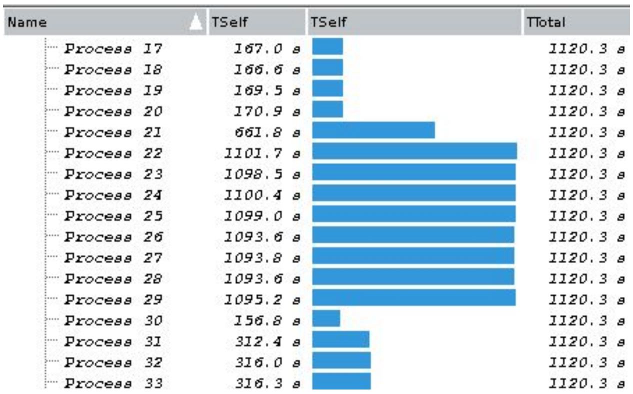


**Figure 2.** Optimized (MPI3+OpenMP) epiSNP workflow. Red shading indicates the process is multithreaded with OpenMP.

## 3 Code Profiling

To find computational hotspots in epiSNP, we used several code profiling programs: the Intel Fortran Loop Profiler for the original serial version, and Allinea MAP in conjunction with the Intel Trace Analyzer and Collector for EPISNPmpi. In both the serial and MPI versions, it was revealed that over $99\%$ of the total execution time was spent in one subroutine called two_snp_test(). The EPISNPmpi workflow is presented in Figure 1. For comparison, the optimizations described in the subsequent sections result in the workflow illustrated in Figure 2.

Four bottlenecks in two_snp_test() accounted for over $75\%$ of the total run time. This included loops containing conditional statements, as well as an $N \times 5$ matrix multiplication of the form $X^T X$. With Trace Analyzer, we discovered a severe load imbalance between MPI processes in EPISNPmpi as seen in Figure 3. This figure shows some MPI processes running three to five times as long as others. These code profiling results guided our serial optimization efforts and revealed that parallel performance would benefit from an improved load balancing method. Importantly, profiling also revealed where *not* to focus our efforts. For example, EPISNPmpi parallelized the single locus tests; however, profiling revealed that this was less than 1% of execution time. We considered the serial performance of the single locus tests to be sufficient after implementing some of the serial optimizations described in Section 5.



| Name | TSelf | TSelf | TTotal |
|---|---|---|---|
| Process 17 | 167.0 s | | 1120.3 s |
| Process 18 | 166.6 s | | 1120.3 s |
| Process 19 | 169.5 s | | 1120.3 s |
| Process 20 | 170.9 s | | 1120.3 s |
| Process 21 | 661.8 s | | 1120.3 s |
| Process 22 | 1101.7 s | | 1120.3 s |
| Process 23 | 1098.5 s | | 1120.3 s |
| Process 24 | 1100.4 s | | 1120.3 s |
| Process 25 | 1099.0 s | | 1120.3 s |
| Process 26 | 1093.6 s | | 1120.3 s |
| Process 27 | 1093.8 s | | 1120.3 s |
| Process 28 | 1093.6 s | | 1120.3 s |
| Process 29 | 1095.2 s | | 1120.3 s |
| Process 30 | 156.8 s | | 1120.3 s |
| Process 31 | 312.4 s | | 1120.3 s |
| Process 32 | 316.0 s | | 1120.3 s |
| Process 33 | 316.3 s | | 1120.3 s |

**Figure 3.** Intel Trace Analyzer Load Balance bar graph. The blue bars represent the fraction of run time spent on computation per MPI process (TSelf/TTotal). Here, processes 21-29 are spending significantly more time computing compared to all other processes in a 66 MPI process run of EPISNPmpi.

## 4 Using Accelerators

Accelerators—such as Graphics Processing Units (GPUs), Field-Programmable Gate Arrays (FPGAs), Micron's Automata Processor, and the Intel Xeon Phi coprocessor—are specialized architectures optimized for high-throughput data-parallel computation. In contrast, conventional CPUs are optimized to execute individual (serial) instruction streams with minimal latency. As accelerators offer more compute capacity within a given power envelope, supercomputers will increasingly rely heavily on accelerators to achieve maximal performance within an acceptable power budget. However, a drawback of accelerators compared with conventional CPUs is that the programmer is exposed to more low-level hardware details in order to run a program, and must have a solid understanding of these details to get good performance. Moreover, many accelerators do not support the software-development toolchains for CPUs that programmers may be familiar with, further increasing the learning curve. In addition, data movement between the CPU and the accelerator can become a bottleneck.

epiSNP has little data movement relative to computation, abundant parallelism in computationally-intensive portions of the code, and relatively small per-process memory footprint in our hybrid MPI3+OpenMP version. These characteristics made it a strong candidate for porting to an accelerator.

It would have been a major effort to port the computationally-expensive portions of epiSNP to a GPU. In addition to rewriting code utilizing another programming language or API, relatively large amounts of code would have to be adapted to express the Single Instruction Multiple Thread (SIMT) parallelism required for good performance on a GPU. Such GPU-optimized code would almost certainly not perform well on the host CPU, so both host and GPU versions of the ported code would need to be maintained.

An alternative to GPUs is the Intel Xeon Phi coprocessor. Utilizing Intel's Many Integrated Core (MIC) architecture, the Intel Xeon Phi is a many-core processor containing around 60 (depending on the model) x86 CPU cores, multiple hardware threads per core, wide SIMD registers (512-bit, vs. 256-bit for current Intel Xeon CPUs), and high memory bandwidth. The peak double-precision floating-point performance of the first-generation MIC (aka Knights Corner) is approximately 1 teraflop/s. The MIC provides a low-development-cost accelerator target for existing CPU-based applications, as most software that runs on the host CPU can run on the MIC (though not necessarily with optimal performance) after a recompilation.

The serial, OpenMP, and MPI optimizations to epiSNP described in the subsequent sections not only substantially improved performance on the Intel Xeon host, but also resulted in good performance on the Intel Xeon Phi coprocessor *without any further code modification*. However, there was a performance bottleneck on the MIC that did not affect the host. This was resolved as described in subsection 5.2.

The resulting application can run on an arbitrary number of host processors and MIC coprocessors using *symmetric mode*, where each MIC functions as an independent node that can execute one or more MPI processes. No code changes were required to accomplish this. It is assumed, however, that the rank 0 MPI process, which performs all I/O and sequential code, runs on a host due to the relatively-slow I/O and single-threaded performance of the MIC.

Our previous epiSNP optimization work (Luecke et al. 2015) employed a static data distribution technique for MPI processes on hosts and MICs. The user was required to manually set an environment variable that influenced the amount of work statically assigned to MPI processes on hosts and MICs (see subsection 7.2). The dynamic load balancing method introduced in this paper obviates this approach.

## 5 Serial Optimizations

Beginning with the original serial epiSNP, we extensively refactored the codebase using modern Fortran constructs for modularity, clarity, and performance. This resulted in the reduction of lines of source code by about $\frac{1}{3}$. We also implemented a number of serial optimizations to address the serial application performance bottlenecks identified using the methods described in section 3. Several of these serial optimizations are described in this section.

### 5.1 Data Type Changes

Compared with double precision, single-precision values consume half the memory/storage, and double the number of data values that can be represented in the same area of cache.

On many architectures (including x86), SIMD instructions can operate on twice as many single-precision elements in a vector register. Because it is beneficial to performance to use single precision where it suffices instead of double precision, epiSNP was modified to represent phenotype values in single precision by default (customizable at compile-time). For the data set described in section 8, the use of mixed precision resulted in only minor numerical differences in some of the reported values, and no differences in the reported SNP pairs with the most-significant epistatic interactions.

epiSNP stored the SNP genotypes of each individual in an array of type default integer (generally 4 bytes). This greatly exceeds the minimum storage (2 bits) required to represent a biallelic SNP. To reduce memory usage and allow more efficient cache utilization, we chose to represent SNPs using int8 (1-byte integer).

## 5.2 Data Structure Changes

epiSNP reports a user-specified number of most-significant epistasis effects. The relevant values (snp1, snp2, trait, effect type, and epistasis effect) are stored in several arrays, sorted by effect. With EPISNPmpi, each MPI process maintains a local copy of these arrays, and the results are merged on the rank 0 process. An $\mathcal{O}(n^2)$ binary insertion sort is used to manage the lists. When an epistasis effect is found that belonged in the lists (i.e., more significant than the least-significant effect stored in the epistasis effect array), a binary search is done to determine the location in the epistasis effect array. The corresponding values are then inserted into the arrays by shifting the other elements to make space before assigning the values to the correct locations in the arrays. In addition to requiring potentially many operations to perform the element shifting, the cache lines of the elements involved in the shift are invalidated.

While this was not a significant bottleneck on the host for the array sizes specified in our benchmark, the insertion routine had worse performance on the Xeon Phi. To reduce the number of operations required to insert an epistasis effect, the five separate arrays were coalesced into a single array of structures. Since one such list is maintained per process, updates to the list must be serialized (via an OpenMP **critical** region) when a process is multi-threaded. On the Xeon Phi, this critical region was a bottleneck: one of the 240 threads updating the list in the critical region could measurably delay access to the critical region for any other threads attempting to update the list.

It was recognized that the problem of maintaining a fixed-size list of the most significant results is analogous to a priority queue where the element with the highest ”priority”—and thus the one to be ”dequeued” next—is the element with largest (least-significant) epistasis effect value. A priority queue is commonly represented using a binary heap data structure. This is a special kind of binary tree that has the *heap property*, where (for a *max-heap*) the value of every node is less than or equal to the value of its parent. The heap can be backed by an array instead of a binary tree. In this case, the left child of the node at index $i$ is located at index $2i$, and the right child at index $2i+1$. Insertions and removals are accomplished in $\mathcal{O}(\log n)$ time, giving the insertion and removal of $n$ elements a time complexity of $\mathcal{O}(n \log n)$. A max-heap implementation from *Introduction to Algorithms, Third Edition* (Cormen et al. 2009) was adapted for this task. On the Xeon Phi, this improved list insertion performance and markedly reduced the maximum cumulative times spent waiting to enter the `critical` region among all OpenMP threads. This maintains the convenience of byte addressability, and avoids any computational expense associated with unpacking an array of 2-bit values before use.

## 5.3 Dead Code Elimination

Approximately 45% of the total execution time of epiSNP was spent in a single region of *dead code*. The results calculated in this region did not contribute to the output, but were stored in the element of an array whose other elements were later utilized. The identification of this region of dead code was achieved only through careful visual inspection of the source code, as the Intel compiler was unable to identify this region during the dead-code elimination phase.

## 5.4 Loop Optimizations

A significant fraction of the epiSNP execution time was spent executing a loop containing nested multi-branch IF constructs. The outer IF construct tests for missing phenotype values, while the inner IF construct updates elements of several arrays based on the SNP genotypes being compared. Branching within a loop inhibits vectorization, and poses a particular performance problem for the ”Knights Corner” generation of the Intel Xeon Phi, which lacks branch prediction hardware.

Listing 1: Excerpt of the original two_snp_test subroutine

```
subroutine two_snp_test(M, residual, snp1, snp2, &
                        missing_trait,...)
integer, intent(in) :: M, snp1(M), snp2(M)
integer, intent(in) :: missing_trait
double precision, intent(in) :: residual(M)
double precision :: b(9,1),df,e(M),msr,x(M,9)
integer :: freq(3,3)
x = 0.0d0; b = 0.0d0; freq = 0; e = 0.0d0
do i=1, M ! for each individual
  if (residual(i) /= missing_trait) then
    if (snp1(i)==0 .and. snp2(i)==0) then
        freq(1,1)=freq(1,1)+1
        x(i,1)=1.0d0
        b(1,1)=b(1,1)+residual(i)
...
    else if (snp1(i)==2 .and. snp2(i)==2) then
      freq(3,3)=freq(3,3)+1
      x(i,9)=1.0d0
      b(9,1)=b(9,1)+residual(i)
    end if
  end if
end do
...
do i=1, M
  if (x(i,1) == 1) then
    e(i)=residual(i)-b(1,1)
...
  else if (x(i,9) == 1) then
    e(i)=residual(i)-b(9,1)
  end if
end do
msr = SUM(e**2)/df
```

To eliminate IF constructs from within these loops, an array containing the indices of the array elements to modify was created. This array is computed in three steps. First, all SNP genotype values representing missing data (i.e., those $> 2$) are assigned a value of 16 (i.e., the 5th bit is set). In the second step, a Fortran MERGE intrinsic function substitutes bit-shifted SNP genotype values (occupying bits 3 and 4) where phenotype data for the current trait exists for that individual, and the missing-data value where it doesn't. The first two steps are done once for each list of individual genotype values for a given SNP, and is thus performed $N$ times. In the third step, the first array is bitwise-ORed the SNP genotype values being compared against (occupying bits 1 and 2 for non-missing data). This bitwise-OR operation, which is performed $\frac{N(N-1)}{2}$ times, can be vectorized by a vectorizing compiler.

Though the resulting loop still doesn't vectorize due to the use of indirect references, the reduction of branching within the loop improved performance. The addition of the Intel-compiler-specific `UNROLL` directive, which signals the compiler to unroll the loop the maximum number of iterations allowed by the Intel compiler, was empirically shown to benefit performance.

Listing 2: Optimized two_snp_test

```
integer(kind=int8), parameter :: MISSING = 16
! valid snps in {0,1,2}
where(snps > 2) snps = MISSING

do s1 = 1, N-1
...
  snps1=MERGE(SHIFTL(snps(:,s1), 2), MISSING, &
        snps(:,s1) /= MISSING .and. &
        residual(:,trait) /= missing_trait)
  do s2 = s1+1, N
...
    call two_snp_test(...)
...
subroutine two_snp_test(...)
...
integer(kind=int8), intent(in) :: snp1(N), snp2(N)
! These arrays were rank 2 in the original
! code for compatibility with matrix operations
! that were removed during dead-code removal.
double precision :: b(0:24)
integer :: freq(0:24)
! The storage size of x is now 1/72 of the
! previous size, enabling it to fit within cache
integer(kind=int8) :: x(N)
...
x = IOR(snp1, snp2)
!DIR$ UNROLL(255)
do i = 1, N
  freq(x(i)) = freq(x(i)) + 1
  b(x(i)) = b(x(i)) + residual(i)
end do
...
msr=SUM((residual-b(x))**2, MASK = x < MISSING)/df
```

However, the Intel Loop Profiler revealed that approximately half of the serial run time is still spent in the loop marked with the `UNROLL` directive. This type of ”histogram” computation (using indirect array addressing) cannot be vectorized using existing x86 or MIC SIMD instruction sets due to the possibility of duplicate index values. The AVX-512 conflict detection instruction set—forthcoming as of this writing, to appear first in the ”Knights Landing” Intel Xeon Phi coprocessor—is designed to allow the detection of and vectorization using conflict-free subsets of index elements (Newburn, CJ 2015). This vector instruction set holds promise beyond wider vector widths for the potential to substantially impact the performance of epiSNP.

### 5.5 DCDFLIB

epiSNP uses DCDFLIB (Double precision Cumulative Distribution Function LIBrary), a Fortran 77 library of routines for computing cumulative distribution functions, inverses, and parameters for various statistical distributions in double precision (Brown et al. 1994b). The `two_snp_test()` subroutine in epiSNP called `cdft()` to calculate the cumulative distribution function and inverse distribution function of the Student's t-distribution. We noticed that `cdft()` could be safely substituted with `cumt()`, which is internally called by `cdft()` given the actual arguments used in epiSNP. Further analysis of the DCDFLIB source code revealed that `cumt()` was a *pure* subroutine (i.e., no side effects), unlike `cdft`, which performs I/O (emits error messages) under certain conditions. An interface block that declared `cumt()` as ELEMENTAL (which implies pure) was created. This served two purposes: 1) to allow the compiler to perform argument type checking for this legacy Fortran 77 subroutine, and 2) to allow the compiler to perform more aggressive optimizations that wouldn't be permissible with an impure subroutine.

The aforementioned serial optimizations resulted in a 12X speedup vs. the original serial epiSNP on the benchmark data set described in Section 8. Approximately 15% of the optimized epiSNP's serial run time is spent executing DCDFLIB subroutines. A subsequent Fortran 90 version, CDFLIB (Brown et al. 1994a)[†], performed slightly better than original Fortran 77 DCDFLIB; however, there were minor numerical differences in the reported epistasis effect p-values. CDFLIB90 (Brown et al. 2002) is a more recent version, coded in a modular style using Fortran 95 language features, that claims to offer improved performance. However, substituting the corresponding CDFLIB90 subroutines in epiSNP revealed that they offered significantly worse performance. Better-optimized versions of these statistical routines would benefit epiSNP and other applications.

## 6 Shared-Memory Parallelization

Shared-memory parallelism is becoming increasingly necessary to exploit the increasing amount of on-chip parallelism found in newer generations of multi-core and many-core processors. The most common API for expressing shared-memory parallelism in scientific applications for multi-core processors and the Xeon Phi coprocessor is OpenMP. Shared-memory parallelism within an MPI process or coarray image is beneficial in several ways. Only one multi-threaded process/image is required to utilize the computational capability within a node (though one per NUMA domain provides best performance for epiSNP and many other applications). Large data structures needing to be replicated in each process/image can be shared among OpenMP threads within the process/image, reducing both

[†] Not to be confused with the original single-precision Fortran 77 CDFLIB (Brown and Lovato 1993)

inter-process communication and per-node memory footprint, and more-efficiently utilizing processor cache. This is especially important for the Intel Xeon Phi, as memory constraints make it impractical to execute enough MPI processes or coarray images on the MIC to fully utilize its computational capacity, and minimization of communication over the slow PCIe bus is desirable. In addition, the OpenMP runtime can dynamically assign loop iterations to threads, facilitating better load balancing within a node. While Fortran 2008 offers a `DO CONCURRENT` construct for parallelizing loops, it could not be used for epiSNP, as there is no facility for mutual exclusion within a `DO CONCURRENT` loop.

### 6.1 Loop Parallelization

When compiled as a pure OpenMP application (without MPI or CAF support), OpenMP parallelizes the outer of the nested loops in which `two_snp_test()` is called. The OpenMP `collapse` clause could not be used in this case, as the inner loop bounds are dependent on the outer-loop index. Dynamic loop scheduling is used, as earlier outer-loop iterations execute more inner-loop iterations.

A max-heap array of structures, `largest_effects`, stores the list of $S$ most-significant epistasis effects and associated metadata (where $M$ is a user-specified parameter). This list is shared between all OpenMP threads in the process/image, reducing both memory usage and the number of list merges needed to provide the final list of $S$ most significant effects. A drawback of this memory-efficient approach is that updates to the list must be serialized within a critical region. Furthermore, the comparison and list update must be performed atomically, so the comparison must also be performed within the critical region. The overhead of having a critical region within each of the $\frac{N(N-1)}{2}$ inner-loop iterations motivated the following optimization.

Before exiting the critical region, the executing thread atomically updates a shared variable containing the least-significant effect stored in the `largest_effects` list (see Listing 3). Before entering the critical region, a thread atomically reads the shared variable into a private variable, and compares the this with a vector of 4 effect values computed in `two_snp_test()`. If any of the computed effects is more significant, the critical region is entered. Because the atomic read may have occurred after a (different) thread in the critical region updated the list, but before updating the shared variable, the comparison must be done again after a thread enters the critical region. As program execution proceeds, the list contains increasingly-significant effects, and the probability of entering the critical region decreases. This method eliminates the vast majority of critical region entrances that would otherwise occur. OpenMP atomic operations are generally implemented as atomic CPU instructions, and present less overhead than OpenMP critical regions.

Listing 3: (pure) OpenMP parallelization

```
type(effect_t) :: largest_effects(S)
!$omp parallel do private(effects, &
!$omp least_effect_private) schedule(dynamic,1)
do snp1 = 1, N-1
  ...
  do snp2 = snp1+1, N
    ...
    call two_snp_test(...)
    ...
!$omp atomic read
!$  least_effect_private = least_effect
!$  if (ANY(effects<least_effect_private)) then
!$omp critical
    do i=1,4
      if (effects(i) < largest_effects(1)%eff) then
        call save_effect(effects(i),...)
      end if
    end do
!$omp atomic write
!$  least_effect = largest_effects(1)%eff
!$omp end critical
    end if
  end do
end do
```

### 6.2 Parallel I/O

For the data set described in Section 8, approximately 2.4G of SNP genotype data organized into 30 files must be read, parsed, and stored in an array. A significant fraction (over 90%) of the execution time of the sequential part (outside the `two_snp_test()` nested loop) was spent performing this task. It was recognized that the I/O bandwidth capacity of the Lustre file system was underutilized.

Minor code changes allowed the sequential loop in which these files were read to be parallelized with OpenMP. Using dynamic loop scheduling with a chunk size of 1, each file was read by a separate thread in a team of threads. This technique lessened the time to read the SNP genotype data by over 80%.

## 7 Distributed-Memory Load Balancing

EPISNPmpi uses a data distribution scheme that requires the number of MPI processes to be a triangular number greater than 2; i.e., a number that is equal to $\frac{N(N+1)}{2}$ for some positive integer $N > 1$. Each (single-threaded) MPI process can utilize only one processor core/thread. Thus, if the aggregate number of processor cores/threads among all compute nodes involved in the computation is not a triangular number, not all of their aggregate computational capability will be utilized.

Building on our serial and shared-memory parallel (OpenMP) optimizations to epiSNP, we created new distributed-memory parallel MPI and CAF versions. These versions allow execution using an arbitrary number of MPI ranks or coarray images, and provide significantly-improved load balancing between ranks/images.

### 7.1 Static Load Balancing with MPI

A pairwise comparison of $N$ elements requires $\binom{N}{2} = \frac{N(N-1)}{2}$ comparisons. This can be implemented in serial code using nested loops; e.g.:

```
do snp1 = 1, N-1
  do snp2 = snp1+1, N
  ! detect epistasis between snp1 and snp2
  end do
end do
```

A simple way to statically distribute work to $P$ MPI processes is to assign iterations of the outer loop to the MPI processes in a round-robin fashion.

```
call MPI_Comm_rank(comm, r, ierror)
call MPI_Comm_size(comm, P, ierror)

do snp1 = r+1, N-1, P
  do snp2 = snp1+1, N
  ! detect epistasis between snp1 and snp2
  end do
end do
```

However, this results in load imbalance: lower-ranked MPI processes do more work than higher-ranked MPI processes (for each block of $P$ iterations, rank $r \in \{0, \ldots, P-1\}$ executes one more iteration of the inner loop than rank $r+1$).

Our original epiSNP optimization effort, described in (Luecke et al. 2015), statically assigned outer-loop iterations to MPI processes such that each process performed similar amount of work.

## 7.2 Dynamic Load Balancing

The time to execute a pairwise locus comparison can vary. When more MPI processes are used for a given problem size, each process executes fewer pairwise comparisons. With static data distribution, this increases the probability that there will exist MPI processes whose execution time more-significantly diverges from the mean execution time. It is important to minimize this load imbalance, as the run time of the ”laggard” MPI processes determines the overall run time of the pairwise-comparison nested loop.

In addition, host CPUs and Xeon Phi coprocessors have different performance characteristics. Our original statically-load-balanced epiSNP required the user to estimate the performance epiSNP on a Xeon Phi relative to a (single) host CPU. This estimate (the *MIC performance factor*) would then cause the number of pairwise comparisons assigned to host and accelerator to differ depending on their respective quantities and performance factor. The MIC performance factor would typically be determined empirically via a trial run of epiSNP on the target system with a small data set. This is obviously inconvenient. It was apparent that a low-overhead dynamic load balancing technique could benefit both performance and usability.

A conceptually straightforward approach for achieving dynamic load balancing is to dynamically load balance the outer loop iterations between processes, while retaining the OpenMP-based dynamic load balancing of the inner loop. It was recognized that the Fortran language already contained the necessary constructs to simply and clearly implement the dynamic load balancing of the outer loop.

### 7.2.1 Dynamic Load Balancing with Coarrays

Coarrays are a feature of the Fortran 2008 standard that allows Single Program Multiple Data (SPMD) parallelism to be expressed using a Partitioned Global Address Space (PGAS) parallel programming model. Data locality is expressed via a simple extension to the Fortran array syntax.

epiSNP was modified to use coarrays instead of MPI. Outer loop iterations were dynamically load balanced between coarray *images* (analogous to MPI *ranks*, except the first image is numbered 1 instead of 0). OpenMP was retained for dynamic load balancing inner loop iterations. Listing 4 describes the basic structure of the resulting pairwise-comparison nested loop. The coarrays (the array `snps` and scalar `snp`) are identified by the presence of a codimension ”`[*]`” in their declarations after the variable name. When referenced with an image selector (e.g., ”`snp[1]`”) the coarray on the specified image is accessed. When referenced without an image selector, the local variable on the same image is accessed.

Listing 4: Dynamic load balancing using coarrays+OpenMP

```
integer(kind=int8), allocatable :: snps(:,:)[*]
integer(kind=int64) :: snp[*] ! index into
...                           ! snps(:,snp)
if (THIS_IMAGE() == 1) then
  snp = 1
  SYNC IMAGES(*) ! image 1 syncs w/ all images
else ! other images need only sync with/ image 1
  SYNC IMAGES(1)
end if
...
call CO_BCAST(snps, 1) ! broadcast from image 1
...
!$omp parallel private(i,snp1,snp2) shared(snp,N)
do while (.true.)
  ! a single thread in the image gets the next
  ! SNP (outer-loop iteration)
  !$omp single
  i = AMO_AFADD(snp[1], 1_int64) ! needs int64 args
  ! broadcast to the other threads in the image
  !$omp end single copyprivate(i)
  ! halve the number of times AMO_AFADD() is
  ! executed and make the number of inner-loop
  ! iterations executed consistent by assigning
  ! SNPs i and N-i to the same image
  if (i > N/2 + IAND(N,1)) exit
  do snp1 = i, MAX(N-i,i), MAX(N-2*i,1)
...
    !$omp do schedule(guided)
    do snp2 = snp1+1, N
...
    end do
  ! nowait allows the first thread to finish
  ! to immediately proceed to fetch the next snp
  !$omp end do nowait
...
  end do
...
end do
```

Image 1 first reads the input data from files, and initializes various data structures, including a scalar coarray (`snp`) that acts as an index into the `snps` coarray. Other images must synchronize with image 1 before accessing its coarrays to ensure image 1 is finished setting them. This could be accomplished with a `SYNC ALL` statement, which acts as a barrier for all images (equivalent to `MPI_Barrier()`. However, as communication occurs only between image 0 and all other images, it is not necessary for the other images to synchronize with each other. The `SYNC IMAGES()` statement allows synchronization with a subset of images, and in this case is used to synchronize only between image 0 and images greater than 0.

The collective subroutine `CO_BCAST()`, which in this case broadcasts the `snps` coarray from image 1 to all other images, is the coarray equivalent of `MPI_Bcast()`. This worked around extremely-slow off-node transfer times observed for *get* operations from all other images (i.e., `snps(:,:) = snps(:,:)[1]`) involving allocatable coarrays of type int8. The CO_BCAST() routine is Cray-specific, but the forthcoming Fortran 2015 standard will include a similar routine (CO_BROADCAST()).

The function AMO_AFADD() (atomic memory operation/atomic fetch and add) is used to fetch the value of the snp coarray from image 1 into local variable i, then increment the value and store it in the snp coarray on image 1. This function is globally atomic with respect to other images, and can thus be safely invoked with identical coarray arguments on multiple images concurrently. Cray interconnects (such as the Aries) contain hardware support for remote atomic memory operations such as this to make them more efficient Alverson et al. (2012). For non-Cray Fortran implementations, the semantics of this operation can be represented (less succinctly) in standard Fortran 2008 using a CRITICAL section:

```
CRITICAL
   k = snp[1]
   snp[1] = k+1
END CRITICAL
```

AMO_AFADD() is Cray-specific, but the Fortran 2015 standard will include the equivalent subroutine ATOMIC_FETCH_ADD().

To quantify the load imbalance present in our previous statically-load-balanced MPI+OpenMP version, the *percent load imbalance* was measured for the pairwise-comparison nested loop. This is defined as the difference between the maximum and minimum run times among all processes/images, expressed as a percent of the maximum run time ($(max - min)/max \times 100$). The percent load imbalance gives an upper bound on the potential for performance gains from load balancing improvements. On 512 nodes of NERSC's Edison supercomputer,[‡] the MPI+OpenMP had a load imbalance of 6.51%, while the CAF+OpenMP version had a load imbalance of 0.58%.

### 7.2.2 Dynamic Load Balancing with MPI-3 RMA

To allow dynamic load balancing on non-Cray systems, we reimplemented dynamic load balancing using MPI-3 RMA operations (Listing 5). This was done due to problems when using CAF with the current versions of Intel and GNU compilers.

CAF was originally designed and implemented by Cray. On Cray systems, CAF programs utilize the proprietary DMAPP one-sided communication library, which is tuned for Cray interconnects. While the CAF implementation of epiSNP achieves improved load balancing on Edison, it may not port to non-Cray systems.

The Intel 15.0.1 Fortran compiler implements coarrays using the Intel MPI library as the underlying transport. MPI-3 passive target communication is used to emulate coarray reads and writes to remote images. However, the Intel compiler does not support the access of remote coarray images from within an OpenMP parallel region. In addition, the Intel MPI library requires that each image have a progress thread (via the MPICH_ASYNC_PROGRESS environment variable) to guarantee progress of passive target communication if the target image is involved in computation. This essentially dedicates one processor core (assuming no Hyper-Threading) to communication for each coarray image, reducing the number of processor cores dedicated to computation.

The GNU Fortran (gfortran) compiler requires the nascent OpenCoarrays library for CAF programs that run on multiple nodes. While OpenCoarrays shows promising performance potential (Fanfarillo et al. 2014), the authors were unable to successfully deploy it on their local cluster for evaluation.

The CAF code was translated to MPI in the following manner. In CAF, a scalar coarray accessible by other images can be declared with integer :: snp[*]. In MPI, the scalar variable must be exposed to remote processes from within a *window* of memory, which defines a region of memory accessible to other processes via MPI RMA operations. Window creation and memory allocation can be performed simultaneously within the collective *MPI_Win_allocate()* routine. This routine requires the window size in bytes as an argument. MPI_Sizeof() is used to determine the size in bytes of a scalar default integer.[§] The accumulate_ops info argument is set to same_op to inform the MPI implementation that all concurrent MPI one-sided accumulate operations (in this case, MPI_Fetch_and_op(...,MPI_SUM,...)) on the same target address will be the same operation. This can obviate the need for the MPI implementation to enforce mutual exclusion on the target address in cases where the hardware can perform those operations atomically. Finally, the resulting allocated memory, which is pointed to by a C pointer, is associated with a Fortran pointer via the Fortran intrinsic function C_F_POINTER().

The semantics of AMO_AFADD() were implemented using MPI_Fetch_and_op(). MPI-3 passive target communication (MPI_Win_lock() and MPI_Win_unlock()) was used so as to not require the target MPI process (rank 0) to be explicitly involved in the communication.

Listing 5: Dynamic Load Balancing using MPI-3 RMA and OpenMP

```
integer(kind=int8), allocatable :: snps(:,:)
integer, pointer :: snp
type(c_ptr) :: p_snp
integer :: i, ierr, info, integer_size, win, ierr
integer, parameter :: one = 1
...
call MPI_Sizeof(i, integer_size, ierr)
call MPI_Info_create(info, ierr)
call MPI_Info_set(info, "accumulate_ops", &
                  "same_op", ierr)
call MPI_Win_allocate( &
     INT(integer_size, kind=MPI_ADDRESS_KIND), &
     1, info, MPI_COMM_WORLD, p_snp, win,ierr)
call C_F_POINTER(p_snp, snp)
snp = 1
...
! broadcast snp genotypes and other relevant data
call MPI_Bcast(snps, N, MPI_INTEGER, 0, &
               MPI_COMM_WORLD, ierr)
...
!$omp parallel private(i,snp1,snp2) shared(snp,N)
do while (.true.)
```

[‡] https://www.nersc.gov/users/computational-systems/edison/

[§] Fortran 2008 added a *C_SIZEOF()* intrinsic function that serves a similar purpose.

```
! a single thread in the MPI process gets the
! next SNP (outer-loop iteration)
!$omp single
  call MPI_Win_lock(MPI_LOCK_EXCLUSIVE, 0, 0, &
                    win, ierr)
  call MPI_Fetch_and_op(one, i, MPI_INTEGER, 0, &
       0_MPI_ADDRESS_KIND, MPI_SUM, win, ierr)
  call MPI_Win_unlock(0, win, ierr)
  ! broadcast to the other threads in the image
  !$omp end single copyprivate(i)
  if (i > N/2 + IAND(N,1)) exit
  do snp1 = i, MAX(N-i,i), MAX(N-2*i,1)
...
    !$omp do schedule(guided)
    do snp2 = snp1+1, N
...
    end do
  !$omp end do nowait
...
  end do
...
end do
```

## 8 Benchmarking

epiSNP was benchmarked on a large bovine genotype and phenotype data set at the Texas Advanced Computing Center (TACC) and the National Energy Research Scientific Computing Center (NERSC). A total of 1,634 Angus sired cattle were genotyped with the Bovine SNP50 Infinium II BeadChip (Illumina, Inc. 2012). These animals' genotypes were imputed to 774,660 SNPs using additional animals within the pedigree genotyped with the Bovine HD BeadChip (Illumina, Inc. 2015). This data set was phenotyped for more than 100 growth, mineral and fatty acid traits. Carcass contemporary group and mineral group were fit as non-genetic factors. As execution time scales linearly with the number of traits, one of the measured phenotypes, stearic acid, was chosen for benchmarking.

### 8.1 Stampede

The TACC Stampede system[¶] was used to benchmark the EPISNPmpi and MPI3+OpenMP epiSNP. Each compute node of Stampede has two 8-core Intel Xeon E5-2680 processors (host) and one or two Intel Xeon Phi SE10P Coprocessors (MIC). The nodes are connected via Mellanox FDR InfiniBand interconnect. Three parallel Lustre file systems are used for storage. The MPI3+OpenMP epiSNP was compiled with Intel Fortran 15.0.2.164 Build 20150121 and run with Intel MPI Version 5.0 Update 2 Build 20141030. Due to limited allocation we did not rerun the original EPISNPmpi program for this paper. Instead we include timing results from the previous paper (Luecke et al. 2015) when EPISNPmpi was compiled with Intel compiler Version 14.0.1.106 Build 20131008 and run using Intel MPI Library Version 4.1 Update 1 Build 20130522.

We benchmarked the MPI3+OpenMP epiSNP using 1) only the host processors, 2) the host processors along with one MIC per node, and 3) the host processors along with two MICs per node. We ran two MPI processes per host (each starting 8 OpenMP threads) and one MPI process per MIC (each starting 240 OpenMP threads). The MPI3+OpenMP epiSNP was compiled using the following compiler options: -ipo -Ofast -xHost. This is the equivalent to -fast without -static, which results in a link error on CentOS 6. [‖] EPISNPmpi performed worse when compiled with -Ofast, thus we used only -ipo and -xHost options for this program.

To run MPI3+OpenMP epiSNP on Stampede we had to move the RMA window accessed by all MPI processes from rank 0 to rank 1, and substitute the fetch-and-increment operation with a get operation on the rank 1 process. Otherwise this process would execute many more iterations than other processes. These modifications are included in the single codebase, and can be enabled at compile-time using preprocessor directives. To utilize more than half of the cores on the first node, we had to modify the default affinity and number of OpenMP threads on this node: the rank 0 process started 15 OpenMP threads, while the rank 1 process started only 1 thread. The following was suggested by XSEDE support staff. When running MPI3+OpenMP epiSNP only on the host processes we set `OMP_PROC_BIND=true` and `OMP_PLACES=cores`. On the first node we also set `I_MPI_PIN_DOMAIN=node`. When running MPI+OpenMP epiSNP with 1 or 2 MICs per node in symmetric mode, `numactl` physcpubind, cpunodebind and membind options were used. `I_MPI_PIN` was set to 0, while `MIC_I_MPI_PIN` was set to 1.

When running MPI3+OpenMP epiSNP with 2 MICs using the aforementioned custom affinity options, we noticed the interesting behavior illustrated in Figure 5. When more than two nodes were used, the number of outer-loop iterations (i.e., number of `MPI_Fetch_and_op() calls invoked` computed by the host processes on every third node of the first 3/4 of the nodes assigned to the job was half number of iterations computed by the host processes on other nodes. A corresponding increase in the time spent blocked waiting for the (`MPI_Win_lock(), MPI_Fetch_and_op(), MPI_Win_unlock()`) sequence to complete was recorded in the affected MPI processes, indicating a communication bottleneck. We did not determine the origin of this issue, but its resolution would improve load balancing and reduce run time. We did not observe this behavior for either MPI3+OpenMP 1 MIC with the custom affinity options, or MPI3+OpenMP 2 MIC without the custom affinity options.

To test scalability we ran benchmarks on different numbers of nodes. Ideally the number of nodes would be a power of two. However since the EPISNPmpi requires the number of MPI processes to be a triangular number, we used 33 nodes instead of 32, 65 nodes instead of 64, and 126 nodes instead of 128. Thus, the tests were run for the following number of nodes: 2, 4, 8, 16, 33, 65, 126. We did not test using larger node counts, as jobs submitted to the normal-2mic queue are limited to 128 nodes. We were not able to run EPISNPmpi on the smaller number of nodes due to queue time limits (the longest queue is 48 hours), thus this version was run only for the 8, 16, 33, 65, and 126 nodes (120, 253, 528, 1035 and 2016 MPI processes).

The MPI3+OpenMP epiSNP was run 3 times, and the averages of the three runs are reported in Table 1. For all

[¶] https://portal.tacc.utexas.edu/user-guides/stampede

[‖] https://software.intel.com/en-us/articles/error-ld-cannot-find-lm

**Table 1.** Run time in minutes (Stampede)

| nodes | EPISNPmpi | MPI3 + OpenMP | MPI3 + OpenMP 1 MIC | MPI3 + OpenMP 2 MIC |
|---|---|---|---|---|
| 1 | N/A | 1421.89 | 767.63 | 505.01 |
| 2 | N/A | 659.90 | 361.19 | 250.05 |
| 4 | N/A | 325.97 | 178.10 | 129.88 |
| 8 | 2338.34 | 158.91 | 88.94 | 65.01 |
| 16 | 1360.06[1] | 79.12 | 44.50 | 32.70 |
| 33 | 649.93 | 38.44 | 21.82 | 16.12 |
| 65 | 335.68 | 19.79 | 11.39 | 8.58 |
| 126 | 185.00 | 10.42 | 6.25 | 4.81 |

[1] used mvapich2/2.0b instead of impi/4.1.1.036

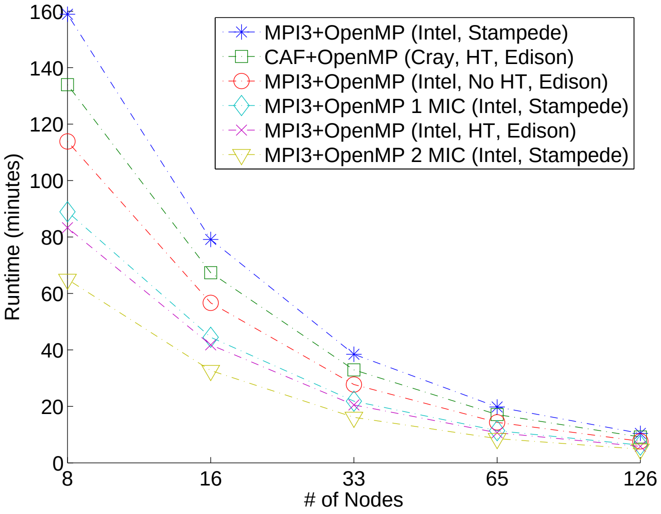


**Figure 4.** Run time in minutes (from 8 to 126 nodes with Cray and Intel compilers on Edison and Stampede)

runs we report the time of the TACC `ibrun` command used on Stampede to launch the processes on all nodes. When running the MPI3+OpenMP epiSNP on 1 node without MICs, the runtime was unexpectedly highly variable (two runs took approximately 23 hours, while the third took approximately 25 hours). The timings for other node counts showed relatively little variation.

Table 1 shows that the runtime of MPI3+OpenMP 2 MIC is less than half of the runtime of MPI3+OpenMP. This is because a MIC coprocessor is approximately 1.5X faster than a host processor.

Table 2 gives the node hours required to run all of the different versions of epiSNP. This is an important criterion since most HPC sites charge by node hours. However being able to perform calculation in a timely manner is a critical consideration in deciding how many nodes to use.

Table 3 shows the big improvements in performance of optimized epiSNP over EPISNPmpi.

## 8.2 Edison

The NERSC Edison supercomputer, a Cray XC30, was used to benchmark the MPI epiSNP both with and without Hyper-Threading (HT), as well as the CAF+OpenMP epiSNP.

**Table 2.** Node hours (Stampede)

| nodes | EPISNPmpi | MPI3 + OpenMP | MPI3 + OpenMP 1 MIC | MPI3 + OpenMP 2 MIC |
|---|---|---|---|---|
| 1 | N/A | 23.70 | 12.79 | 8.42 |
| 2 | N/A | 22.00 | 12.04 | 8.34 |
| 4 | N/A | 21.73 | 11.87 | 8.66 |
| 8 | 311.78 | 21.19 | 11.86 | 8.67 |
| 16 | 362.68 | 21.10 | 11.87 | 8.72 |
| 33 | 357.46 | 21.14 | 12.00 | 8.87 |
| 65 | 363.65 | 21.44 | 12.34 | 9.29 |
| 126 | 388.50 | 21.89 | 13.13 | 10.10 |

**Table 3.** Speedup over EPISNPmpi (Stampede)

| nodes | MPI3 + OpenMP | MPI3 + OpenMP 1 MIC | MPI3 + OpenMP 2 MIC |
|---|---|---|---|
| 8 | 14.72 | 26.29 | 35.97 |
| 16 | 17.19 | 30.57 | 41.59 |
| 33 | 16.91 | 29.78 | 40.32 |
| 65 | 16.96 | 29.47 | 39.14 |
| 126 | 17.76 | 29.58 | 38.43 |

The MPI epiSNP was compiled with the Intel Fortran compiler (version 15.0.1) using the `-fast` option, and statically linked with the DMAPP library. The `MPICH_RMA_OVER_DMAPP` environment variable was set to 1 to use a DMAPP-based version of MPI RMA, as this significantly improved the performance of `MPI_Fetch_and_op()`. The `MPICH_RMA_USE_NETWORK_AMO` environment variable, which enables network atomic memory operations for several MPI routines (including `MPI_Fetch_and_op()`), was not set. This is because setting it resulted in a run-time error from the MPI library.

To more closely map the performance of the MPI epiSNP on Stampede to Edison, epiSNP was first run without HT enabled, using the `aprun` options `--pes-per-node 2 --pes-per-numa-node 1 --cpus-per-pe 12 --CPUs 1 --cpu-binding numa_node`. The `OMP_WAIT_POLICY` environment variable was set to ”active”.

With HT enabled, the MPI epiSNP was run with different numbers of ranks/images using the `aprun` options `--pes-per-node 2 --pes-per-numa-node 1 --cpus-per-pe 24 --CPUs 2 --cpu-binding numa_node`. The `OMP_PLACES` environment variable was set to ”threads”, as the default (”cores”) caused oversubscription from both OpenMP threads executing on a core being pinned to the same hardware thread. The `OMP_WAIT_POLICY` environment variable was not set to ”active” (defaulting to ”passive”), as ”active” caused performance degredation. This may be due to contention with the other thread executing on the same processor core, as an active wait policy causes OpenMP threads to spin while waiting (e.g., to acquire a lock).

The Cray 8.4.0 compiler was used to compile the CAF+OpenMP epiSNP. The MPI3+OpenMP version was

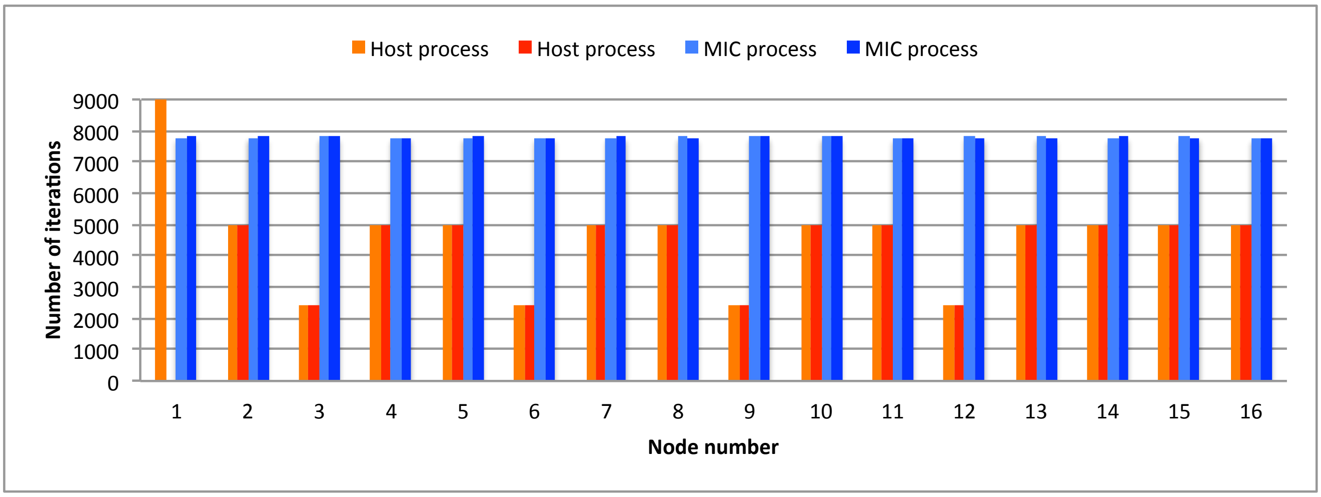


**Figure 5.** Number of `MPI_Fetch_and_op()` calls per MPI rank for MPI3+OpenMP 2 MIC on 16 nodes of Stampede. This is a measure of the amount of work done per process. There are four MPI processes per node: the first two are host processes, while the latter two are MIC processes. The host processes on every third node of the first $3/4$ of the nodes assigned to the job perform approximately half as much work as the host processes on other nodes. The affected processes spend more time blocked in the (`MPI_Win_lock()`,`MPI_Fetch_and_op()`,`MPI_Win_unlock()`) sequence, indicating a communication bottleneck of undetermined origin.

also compiled to compare CAF with MPI using the same compiler. For the CAF+OpenMP version, no compiler options were used. The `XT_SYMMETRIC_HEAP_SIZE` environment variable was set to 2G, as the default was inadequate for storing the SNP genotype data within a coarray on each image. The MPI version was compiled with no compiler options, statically linked with the DMAPP library, and run with the `MPICH_RMA_OVER_DMAPP` environment variable set to 1. Both MPI3+OpenMP and CAF+OpenMP versions were run with HT enabled, using the `aprun` options `--pes-per-node 2 --pes-per-numa-node 1 --cpus-per-pe 24 --CPUs 2` for each node count. The `OMP_PROC_BIND` environment variable was not set, as any value other than ”false” caused all threads in the process/image to be bound to the same core. Adding the `aprun -cc none` option resulted in two OpenMP threads per core, where both were pinned to the same hardware thread. The MPI version was run only once per node count instead of three times due to remaining allocation limitations.

The timing results (Table 4) indicate that the Intel compiler optimized the epiSNP code better than the Cray compiler, as evidenced by the approximately 16% performance improvement of the MPI3+OpenMP epiSNP when compiled with the Intel compiler instead of the Cray compiler.

## 8.3 Comparing Stampede and Edison

Figure 4 graphically summarizes the runtimes of the optimized epiSNP versions listed in Table 1 and Table 4. The MPI3+OpenMP epiSNP was approximately 40% faster on Edison (without HT) vs. Stampede (without MICs). Other HPC applications have observed comparable performance gains when transitioning from similar 8-core ”Sandy Bridge” Intel Xeon processors to similar 12-core versions of the subsequent ”Ivy Bridge” Intel Xeon processors (Deshmukh 2014).

**Table 4.** Run time in minutes (Edison)

| nodes | MPI3 + OpenMP (Cray, HT)[1] | CAF + OpenMP (Cray, HT) | MPI3 + OpenMP (Intel, No HT) | MPI3 + OpenMP (Intel, HT) |
|---|---|---|---|---|
| 1 | 1069.23 | 1064.60 | 895.08 | 657.81 |
| 2 | 535.71 | 532.22 | 455.18 | 328.72 |
| 4 | 267.91 | 267.47 | 224.73 | 165.54 |
| 8 | 134.54 | 133.97 | 113.87 | 83.28 |
| 16 | 68.10 | 67.34 | 56.66 | 41.80 |
| 33 | 33.16 | 32.93 | 27.74 | 20.51 |
| 65 | 17.12 | 17.09 | 14.32 | 10.67 |
| 126 | 9.17 | 9.13 | 7.63 | 5.82 |

[1] Only one trial was run per node count instead of three due to insufficient remaining allocation

Hyper-Threading produced an unexpectedly-large performance benefit for epiSNP on Edison, resulting in an average speedup of 1.35x vs. without HT. Many HPC systems (including Stampede) have HT disabled on the compute nodes at boot via a BIOS setting. This is because HT has little/no benefit for many HPC applications, and is even detrimental to the performance of some. On Edison, HT is enabled, but not used unless the user explicitly requests it via the `aprun --CPUs 2` option. Using HT, the performance of Edison is better than the performance of the Stampede nodes with one MIC, while having better power efficiency. Each of the two Xeon E5-2695 v2 CPUs in an Edison node has a thermal design power (TDP) of 115W, while each of the two E5-2680 CPUs in Stampede node has a TDP of 130W—and the Xeon Phi SE10P adds a 300W TDP.

This experience suggests that it may be beneficial to allow the use of HT if the user explicitly requests it via the resource manager and/or OpenMP (or other) runtime, rather than disallowing the use of HT for all applications.

## 9 Conclusions

The purpose of this work was to build upon our previous optimizations to epiSNP, further improving performance and usability on both host and Xeon Phi coprocessor. We implemented dynamic load balancing in CAF and MPI-3 RMA versions to reduce load imbalance, especially for heterogeneous systems containing both hosts and Xeon Phi coprocessors. As a result, epiSNP can run on multiple nodes, each equipped with an arbitrary number of Xeon Phi coprocessors, without requiring the user to empirically determine a static-load-balancing parameter. In addition, data structure changes reduced thread contention on the Intel Xeon Phi coprocessor.

Preprocessor directives allow six versions of epiSNP (serial, OpenMP, MPI3, CAF, MPI3+OpenMP, and CAF+OpenMP) to be compiled from a single code base. These versions perform substantially better than the original EPISNPmpi. For a large 774,660 SNP data set with 1,634 individuals, our efforts yielded a performance improvement over EPISNPmpi of 17.76X for MPI3+OpenMP, 29.58X for MPI3+OpenMP with 1 MIC, and 38.43X for MPI3+OpenMP with 2 MICs on 126 nodes of TACC's Stampede supercomputer. Benchmarks on the Edison supercomputer revealed that the use of Hyper-Threading resulted in significantly better performance (average 1.35X speedup) than without Hyper-Threading.

The practical application of this work is to scale up to analyze all variants that are segregating in a population. While the resulting optimized epiSNP can manage the 774,660 variant data set size used in this study, scalability challenges remain for even bigger data. For example, the 1000 Bulls Project (Daetwyler et al. 2014) and 1000 Genomes Project (1000 Genomes Project Consortium 2015) have identified 28 and 88 million variants in cattle and humans. Epistatic interaction analyses using all these variants would take approximately 1,300 and 13,000 times longer than the 774,660 variant data set. This means that it would take 126 2-MIC Stampede nodes about 4.5 days to analyze a single bovine trait. To analyze a single human trait would require well over 40 days. The smaller analysis could be done in under a day using 800 Stampede 1-MIC nodes. The larger analysis, however, would take the entire 6400-node Stampede supercomputer about a day to complete. Thus, software, hardware, and/or algorithmic advances are needed in this problem domain to be able to perform the largest analyses within a reasonable amount of time using the statistical methodology applied here.

## Acknowledgment

The authors thank Dr. Yang Da for granting access to the epiSNP source code, and Dossay Oryspayev for his assistance with code optimization.

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

The research reported in this paper is partially supported by the HPC@ISU equipment at Iowa State University, some of which has been purchased through funding provided by NSF under MRI grant number CNS 1229081 and CRI grant number 1205413.

This work was created using resources or cyberinfrastructure provided by the iPlant Collaborative. The iPlant Collaborative is funded by a grant from the National Science Foundation (#DBI-0735191).
URL: `http://www.iplantcollaborative.org`

The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper.
URL: `http://www.tacc.utexas.edu`

This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

### Author Biographies

**Nathan Weeks** received B.S. degrees in Computer Science (2002) Mathematics (2003) from South Dakota State University, and an M.S. (2012) in Computer Science from Iowa State University. He is currently pursuing a Ph.D. in Computer Science at Iowa State University. His research interests include parallel computing, software application optimization, and bioinformatics.

**Glenn R. Luecke** received his B.S. degree from Michigan State University in Mathematics and his Ph.D. in Mathematics from the California Institute of Technology. He is currently Professor of Mathematics, adjunct Professor of Computer Engineering, Senior Member of the ACM, Director of an HPC group and in charge of HPC education and training at Iowa State University. Professor Luecke's HPC group is involved in research in the areas of parallel algorithms, parallel tools, and the evaluation of high performance computing systems. He has had over 60 graduate students, visiting scholars, and post-doctoral students.

**Brandon Groth** is a research assistant in the Department of Mathematics HPC group at Iowa State University. He received his B.S. in Mathematics from the University of Wisconsin-La Crosse in 2012. He is currently seeking a Ph.D. in Applied Mathematics with an emphasis on HPC under Dr. Glenn Luecke. His research interests include scientific computing, mathematical applications in the sciences, and numerical analysis.

**Marina Kraeva** received the B.S. (1991) and M.S. (1993) degrees in Mathematics and Applied Mathematics from Novosibirsk State University, Russia. In 1999 she received her Ph.D. in Computer Science from the State Technical University of Novosibirsk, Russia and joined the High Performance Computing group at Iowa State University. She is interested in performance evaluation of HPC systems and tools and assists ISU faculty and students in program parallelization and optimization.

**Luke Kramer** received his B.S. (2012) in Genetics, Cell Biology, and Development from the University of Minnesota, Twin Cities. Currently he is a Ph.D. student at Iowa State University in the department of Animal Science. His research involves the study of both fatty acid epistasis and response to vaccination in beef cattle.

**James E. Koltes** received a B.S. (2001) in dairy science and genetics from the University of Wisconsin-Madison and a Ph.D. (2007) in genetics from Iowa State University. He joined the faculty of the Department of Animal Science at the University of Arkansas in 2015. His research interests are in genetics, genomics and epigenetics of growth and health traits in livestock species.

**Li Ma** received his B.S. degree (2002) in Mathematics from Fudan University, and his M.S. (2010) in Statistics and Ph.D. (2010) in Quantitative Genetics from the University of Minnesota. He joined the University of Maryland in 2013 as an assistant professor with research focused on population and statistical genetics, bioinformatics, and fast computing tool development for large-scale genetic data analysis.

**James Reecy** received his B.S. (1990) in Animal Science from South Dakota State University, his M.S. (1992) in Animal Science from the University of Missouri, and his Ph.D. (1995) in Animal Science from Purdue University. He was then a Post doctoral fellow at Baylor College of Medicine (1999). In 1999, he joined Iowa State University, where he became a full professor in 2009. He has extensively published in the areas of growth and development, livestock genomics and bioinformatics. He is a coauthor on more than 115 technical papers on these topics.

## References

1000 Genomes Project Consortium (2015) A global reference for human genetic variation. *Nature* 526(7571): 68–74.

Alverson B, Froese E, Kaplan L and Roweth D (2012) Cray XC series network. URL `http://www.cray.com/sites/default/files/resources/CrayXC30Networking.pdf`.

Brown BW and Lovato J (1993) CDFLIB: Library of Fortran Routines for Cumulative Distribution Functions, Inverses, and Other Parameters. URL `http://lib.stat.cmu.edu/general/cdflib`.

Brown BW, Lovato J and Russell K (1994a) CDFLIB. URL `https://people.sc.fsu.edu/~jburkardt/f_src/cdflib/cdflib.html`.

Brown BW, Lovato J and Russell K (1994b) DCDFLIB: Library of Fortran Routines for Cumulative Distribution Functions, Inverses, and Other Parameters. URL `http://netlib.org/random/`.

Brown BW, Venier J and Serachitopol D (2002) CDFLIB90 Fortran 95 routines for cumulative distribution functions, their inverses and more. URL `https://biostatistics.mdanderson.org/SoftwareDownload/SingleSoftware.aspx?Software_Id=21`.

Cormen TH, Leiserson CE, Rivest RL and Stein C (2009) *Introduction to Algorithms, Third Edition*. 3rd edition. The MIT Press. ISBN 0262033844, 9780262033848.

Daetwyler HD, Capitan A, Pausch H, Stothard P, Van Binsbergen R, Brøndum RF, Liao X, Djari A, Rodriguez SC, Grohs C et al. (2014) Whole-genome sequencing of 234 bulls facilitates mapping of monogenic and complex traits in cattle. *Nature genetics* 46(8): 858–865.

Deshmukh M (2014) Comparing Sandy Bridge vs. Ivy Bridge processors for HPC applications. URL `http://dell.to/20EHEUD`.

Fanfarillo A, Burnus T, Cardellini V, Filippone S, Nagle D and Rouson D (2014) OpenCoarrays: Open-source Transport Layers Supporting Coarray Fortran Compilers. In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14. New York, NY, USA: ACM. ISBN 978-1-4503-3247-7, pp. 4:1–4:11. DOI:10.1145/2676870.2676876.

González-Domínguez J, Kässens JC, Wienbrandt L and Schmidt B (2015) Large-scale genome-wide association studies on a GPU cluster using a CUDA-accelerated PGAS programming model. *International Journal of High Performance Computing Applications* : 506–510.

Gonzalez-Dominguez J, Ramos S, Tourino J and Schmidt B (2015) Parallel Pairwise Epistasis Detection on Heterogeneous Computing Architectures. *IEEE Transactions on Parallel and Distributed Systems* PP(99): 1–1. DOI:10.1109/TPDS.2015.2460247.

González-Domínguez J, Wienbrandt L, Kassens J, Ellinghaus D, Schimmler M and Schmidt B (2015) Parallelizing Epistasis Detection in GWAS on FPGA and GPU-Accelerated Computing Systems. *Computational Biology and Bioinformatics, IEEE/ACM Transactions on* 12(5).

Goudey B, Abedini M, Hopper JL, Inouye M, Makalic E, Schmidt DF, Wagner J, Zhou Z, Zobel J and Reumann M (2015) High performance computing enabling exhaustive analysis of higher order single nucleotide polymorphism interaction in genome wide association studies. *Health Information Science and Systems* (Suppl 1): S3.

Illumina, Inc (2012) BovineSNP50 v2 DNA Analysis BeadChip. URL `http://www.illumina.com/content/dam/illumina-marketing/documents/products/datasheets/datasheet_bovine_snp5O.pdf`.

Illumina, Inc (2015) BovineHD DNA Analysis Kit. URL `http://www.illumina.com/content/dam/illumina-marketing/documents/products/datasheets/datasheet_bovineHD.pdf`.

Koesterke L, Stanzione D, Vaughn M, Welch SM, Kusnierczyk W, Yang J, Yeh CT, Nettleton D and Schnable PS (2011) An efficient and scalable implementation of SNP-pair interaction testing for genetic association studies. In: *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*. IEEE, pp. 523–530.

Luecke GR, Weeks NT, Groth BM, Kraeva M, Ma L, Kramer LM, Koltes JE and Reecy JM (2015) Fast Epistasis Detection in Large-Scale GWAS for Intel Xeon Phi Clusters. In: *Trustcom/BigDataSE/ISPA, 2015 IEEE*, volume 3. pp. 228–235. DOI:10.1109/Trustcom.2015.637.

Ma L, Runesha HB, Dvorkin D, Garbe JR and Da Y (2008) Parallel and serial computing tools for testing single-locus and epistatic SNP effects of quantitative traits in genome-wide association studies. *BMC Bioinformatics* 9(1): 315.

Newburn, CJ (2015) Coding for the future: Knights landing and beyond. URL `http://go.usa.gov/c4adP`.

Pütz B, Kam-Thong T, Karbalai N, Altmann A, Müller-Myhsok B et al. (2013) Cost-effective GPU-grid for genome-wide epistasis calculations. *Methods Inf Med* 52(1): 91–95.

Zhou Z, Liu G, Su L, Yan L and Han L (2013) Cchi: An efficient cloud epistasis test model in human genome wide association studies. In: *Biomedical Engineering and Informatics (BMEI), 2013 6th International Conference on*. IEEE, pp. 787–791.