UPC++ for Bioinformatics: A Case Study Using Genome-Wide Association Studies

Jan C. Kässens*, Jorge González-Domínguez†, Lars Wienbrandt*, Bertil Schmidt†

*Department of Computer Science, Christian-Albrechts-University of Kiel
†Parallel and Distributed Architectures Group, Johannes Gutenberg University-Mainz

***Abstract*—Modern genotyping technologies are able to obtain up to a few million genetic markers (such as SNPs) of an individual within a few minutes of time. Detecting epistasis, such as SNP-SNP interactions, in Genome-Wide Association Studies is an important but time-consuming operation since statistical computations have to be performed for each pair of measured markers. Therefore, a variety of HPC architectures have been used to accelerate these studies. In this work we present a parallel approach for multi-core clusters, which is implemented with UPC++ and takes advantage of the features available in the Partitioned Global Address Space and Object Oriented Programming models. Our solution is based on a well-known regression model (used by the popular BOOST tool) to test SNP-pairs interactions. Experimental results show that UPC++ is suitable for parallelizing data-intensive bioinformatics applications on clusters. For instance, it reduces the time to analyze a real-world dataset with more than 500,000 SNPs and 5,000 individuals from several days when using a single core to less than one minute using 512 nodes (12,288 cores) of a Cray XC30 supercomputer.**

***Index Terms*—PGAS, UPC++, GWAS, Bioinformatics**

I. Introduction

High-throughput genotyping technologies allow the collection of hundreds of thousands to a few million genetic markers, such as Single Nucleotide Polymorphisms (SNPs), from individual DNA samples. In Genome-Wide Association Studies (GWAS), these genotypes are typically measured for several thousand individuals and then linked to a given phenotype of each individual, such as the presence (case) or absence (control) of an associated disease. In classical GWAS each genetic marker is analyzed separately in order to identify markers showing significant differences in genotype frequencies between cases and controls. Unfortunately, this approach is generally not powerful enough to model complex traits for which the detection of joint genetic effects (epistasis) needs to be considered [1], [2], [3]. In (2-way) statistical epistasis each pair of measured markers is therefore tested in order to discover significant interactions that explain the given phenotype. Consequently, a number of algorithms have been developed to address the problem of detecting epistasis in recent years [4], [5], [6]. The main goal of these approaches is to find pairs of SNPs whose joint values show a statistically significant difference between cases and controls and thus they provide a list with pairs that could explain a substantial proportion of genetic variation leading to disease.

Computing epistasis is highly time-consuming due to the large number of pairwise tests to be calculated. For example, even for a moderately-sized dataset consisting of 500,000 SNPs there are about 125 billion pairwise interaction tests to be performed. Since both the availability and size of GWAS datasets are increasing rapidly, finding faster solutions is of high importance to research in this area. Thus, recent publications have addressed this problem by using dedicated High Performance Computing (HPC) hardware such as GPUs [7], [8], [9] and FPGAs [10]. **In this work we provide a parallel tool to perform GWAS on multi-core clusters based on regression models**. Our approach is based on Unified Parallel C++ (UPC++) [11], a new extension to C++ that combines the advantages of both Partitioned Global Address Space (PGAS) and the Object Oriented Programming (OOP) paradigms.

The paper is organized as follows. Section II reviews some related work. Necessary background information is provided in Section III (e.g. the characteristics of the UPC++ language and the statistical test applied in the GWAS analyses). Our parallelization approach, as well as the employed optimization techniques, are described in Section IV. Experimental evaluations are presented in Section V. Section VI concludes the paper.

II. Related Work

The development of tools to find epistasis has attracted extensive research interests. Among them, statistical regression methods are very popular to perform GWAS [12], [13], [14]. Other approaches are based on ROC curves [8] or the dependency difference [9]. All these methods calculate computationally expensive measurements for all the SNP-pairs in order to evaluate if they show epistasis.

Many recent approaches are filtration-based; i.e. they first apply a computationally faster filter and subsequently perform the full statistical analysis only to the SNP-pairs not discarded by the preliminary filter. For instance, SNPHarvester [15] uses path algorithms to identify several groups of SNPs associated to the same disease. Then, it applies the statistical method only to the pairs generated within each group. SNPRuler [16] narrows the search space through a learning approach based on predictive rule learning. SIXPAC [17] utilizes a novel randomization technique to minimize the number of tests. BOOST [18] introduces the Kirkwood Superposition Approximation (KSA) as preliminary filter. The authors prove that it is an upper bound for the regression model. Motivated by its high precision and high speed, we have based our UPC++ parallelization on the BOOST approach. Moreover, this tool is widely used in biomedical research to perform their GWAS analyses (see for instance [19], [20], [21]). Furthermore, statistical filters and methods of different tools could also be applied in our UPC++ implementation by merely changing the internal statistical tests, and without modifying the data distribution, optimization techniques or other features related to the parallel implementation.

Previous approaches to accelerate GWAS have been based on a variety of HPC architectures. GBOOST, a CUDA implementation of BOOST, was presented in [7]. GWIS [8], which is based on ROC curves, and iLOCi [9], which is based on dependency differences, also provide GPU implementations. An FPGA implementation of iLOCi was further presented in [10]. **Earlier GWAS parallelizations on HPC clusters using Message Passing Interface (MPI) were developed [22], [23]. However, up to our knowledge, HPC cluster implementations of modern tools to look for epistasis based on the PGAS paradigm do not exist yet.**

We have used UPC++ [11], an extension of C++ for parallel computing, which has evolved from Unified Parallel C (UPC) [24], [25]. PGAS languages (such as UPC, Co-Array Fortran [26] or Titanium [27]) are easier to use than message passing counterparts [28], [29] and can also obtain better performance than them thanks to the one-sided communication [30], [31], [32]. UPC++ combines these advantages of the PGAS model and the OOP paradigm. Due to the novelty, this is also the first work that studies the suitability of UPC++ for parallelizing a big data bioinformatics application.

III. Background

The goal of parallelized 2-way GWAS tools is the exploitation of HPC facilities in order to accelerate the detection of epistasis. They work with datasets that contain information about biallelic genetic markers measured from a large cohort of individuals. Major alleles are denoted with capital letters and minor alleles with lowercase letters. Therefore, for each SNP there are three genotypes $\{AA, Aa, aa\}$, which are numerically represented as $\{0,1,2\}$. Each individual is characterized as case or control, depending on the presence or absence of an associated disease.

Two SNPs present epistasis (or interaction) if their combination leads to significantly higher discrimination between cases and controls in comparison to using any of these two SNPs individually.

A. Contingency Tables, Log-Linear Models and KSA Filter

The number of SNPs and individuals are denoted as M and N, respectively. The individuals are categorized as cases (value 0) and controls (value 1). The filters that identify which SNP-pairs show interaction use a 3x3x2 contingency table per pair. As seen in the example of Table I, each cell $< i, j, k >$ stores the count of individuals categorized as k (case or control) with the value of the first SNP as i, and the second SNP as j. We can also fill the contingency table with probabilities: $\pi_{ijk} = n_{ijk}/N$.

Table I
Example of contingency table

Cases	**SNP2=0**	**SNP2=1**	**SNP2=2**
SNP1=0	n_{000}	n_{010}	n_{020}
SNP1=1	n_{100}	n_{110}	n_{120}
SNP1=2	n_{200}	n_{210}	n_{220}
Controls	**SNP2=0**	**SNP2=1**	**SNP2=2**
SNP1=0	n_{001}	n_{011}	n_{021}
SNP1=1	n_{101}	n_{111}	n_{121}
SNP1=2	n_{201}	n_{211}	n_{221}

As explained in Section II, we use the same tests as in BOOST [18], where Wan et al. prove that the search for interaction with regression models can be simplified using log-linear models. They translate our definition of epistasis to the perspective of the log-linear models as the information contained in the joint distribution but not in its lower-order factorization. This definition leads to measure interaction as $\hat{L}_S - \hat{L}_H$, where $\hat{L}_S$ and $\hat{L}_H$ represent the maximum log-likelihood of the saturated and the homogeneous association models, respectively. It can be calculated from the values of the contingency table as:

$$N \sum_{ijk} \left[\hat{\pi}_{ijk} \log \left(\frac{\hat{\pi}_{ijk}}{\hat{p}_{ijk}} \right) \right]$$

where $\hat{\pi}_{ijk}$ is the joint distribution obtained under the saturated model and $\hat{p}_{ijk}$ the distribution obtained under the homogeneous association model. They establish that all pairs with log-linear measure higher than certain threshold $THRES$ show epistasis. Although this log-linear model is affordable, it still requires a lot of computation as $\hat{p}_{ijk}$ has to be computed by iterative methods. This is the reason why BOOST applies a simpler filter based on the Kirkwood Superposition Approximation (KSA). The authors proved the following upper bound:

$$\begin{aligned}
\hat{L}_{\text{S}} - \hat{L}_{\text{H}} &\leq \hat{L}_{\text{S}} - \hat{L}_{\text{KSA}} \\
\hat{L}_{\text{S}} - \hat{L}_{\text{KSA}} &= N \sum_{ijk} \left[\hat{\pi}_{ijk} \log \left(\frac{\hat{\pi}_{ijk}}{\hat{p}^{k}_{ijk}} \right) \right] \\
\hat{p}^{k}_{ijk} &= \frac{1}{\eta} \frac{\pi_{ij.}\pi_{i.k}\pi_{.jk}}{\pi_{i..}\pi_{.j.}\pi_{..k}} \\
\eta &= \sum_{ijk} \frac{\pi_{ij.}\pi_{i.k}\pi_{.jk}}{\pi_{i..}\pi_{.j.}\pi_{..k}}
\end{aligned}$$

The equations above show that the KSA value can be directly calculated from the cells of the contingency table without iterative methods. Therefore, BOOST accelerates its analysis using the KSA filter ($\hat{L}_S - \hat{L}_{KSA}$). From now, we call the value of $\hat{L}_S - \hat{L}_{KSA}$ for a specific SNP-pair its "KSA value". As the KSA value is an upper bound of the log-linear measure, we calculate it for all SNP-pairs and discard those with a KSA value lower than $THRES$. Finally, we only apply the log-linear filter to the remaining pairs. For simplicity, we refer to [18] to find the proofs and further explanation of the KSA and log-linear filters.

B. The Memory Model in UPC++

The execution model of UPC++ is Single Program Multiple Data (SPMD). As this language is able to work on both shared-memory and distributed-memory systems, each independent execution unit (from now, UPC++ process) can be implemented as an OS process or a POSIX thread (Pthread). UPC++ takes advantage of C++ language features, such as templates, object-oriented design, operator overloading, and lambda functions (in C++ 11) to provide advanced PGAS features.

As all PGAS languages, UPC++ exposes a global shared address space to the user which is logically divided among processes, so each process is associated or presents affinity to a different part of the shared memory. Moreover, UPC++ also provides a private memory space per process for local computations, as shown in Figure 1. Therefore, each process has access to both its private memory and to the whole global memory space (even the parts that do not present affinity to it) with read/write functions. This memory specification combines the advantages of both shared and distributed programming models. On the one hand, the global shared memory space facilitates the development of parallel codes, allowing all processes to directly read and write remote data without explicitly notifying the owner. On the other hand, performance can be increased by taking data affinity into account. Typically, accesses to remote data are more expensive than the accesses to local data (i.e. accesses to private memory and to shared memory with affinity to the process).

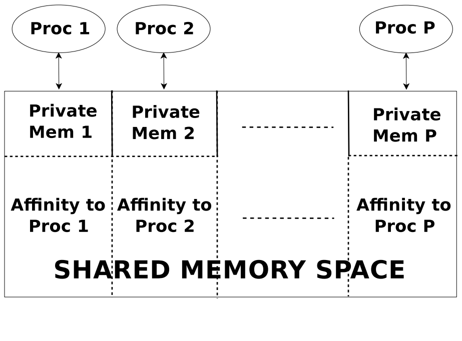

Figure 1. UPC++ memory model

As an extension of the C++ language, UPC++ provides functionality to access memory through pointers. Firstly, regular C++ pointers can access local memory (either private memory or the part of the global memory with affinity to the process). Secondly, UPC++ provides a novel generic `global_ptr` type to reference shared objects in the global address space. A global pointer encapsulates both the process ID and the local address of the shared object referenced by the pointer. Operator overloading is used to provide the semantics of pointers in the global address space. Finally, global pointers can be casted to regular C++ pointers, which then point to the local address of the shared object.

IV. UPC++ Implementation of our GWAS Tool

Due to the symmetry of the tests, only $M(M-1)/2$ SNP-pairs have to be analyzed in a GWAS. In our parallel implementation, each process calculates contingency tables and performs KSA and log-linear tests (explained in Section III-A) to the SNP-pairs assigned to it. Then, it outputs a subset of pairs that pass the log-linear filter. Large-scale GWAS datasets (consisting typically of around 5M SNPs and around 100,000 individuals) often exceed the main memory capacity of an individual compute node. Thus, we have designed the following partitioning scheme. In order to balance the memory usage per node, the biallelic information of the SNPs is distributed among the processes in a block-cyclic way, where the user can specify the number of blocks per process. This biallelic information is stored in shared memory so it can be directly accessed by all processes. For a single SNP, the values for all the individuals have affinity to the same process. As the information is distributed, the UPC++ processes may need information stored in remote memory to analyze whether the SNP-pairs associated to them present interaction. We focus on balancing the workload and minimizing the number of copies from remote memory in order to reduce communication overhead.

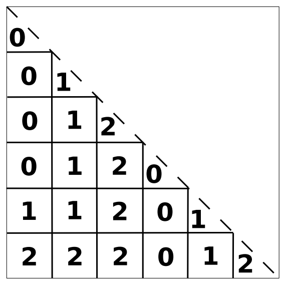

Figure 2. Example of SNP-pairs distribution using 3 processes and 2 blocks per process. Only SNP pairs below the dashed line have to be considered.

Figure 2 illustrates which SNP-pairs are studied by each UPC++ process in an example with 3 processes where the biallelic data is distributed using 2 blocks per process (6 blocks in total). If the cell (i, j) in the matrix is associated to process p, it means that p analyzes the pair formed by the SNPs i and j. In this example there are 21 "metablocks" of SNP-pairs, and each process studies 7 of them. In order to compute the pairs contained in the metablocks just below the diagonal (from now "diagonal-metablocks"), only one block of information is necessary. The distribution assures that the process associated to that diagonal-block is the one that has affinity to the necessary biallelic information (e.g. Process 0 computes the first and the fourth diagonal-metablock in the example of Figure 2). Nevertheless, two blocks of information are necessary in order to analyze the SNP-pairs of the other metablocks. The distribution assures that metablocks are computed by processes that already have, at least, one of the necessary information blocks in their local shared memory. Therefore, in the worst case, only one block of information must be copied from remote memory (for instance,

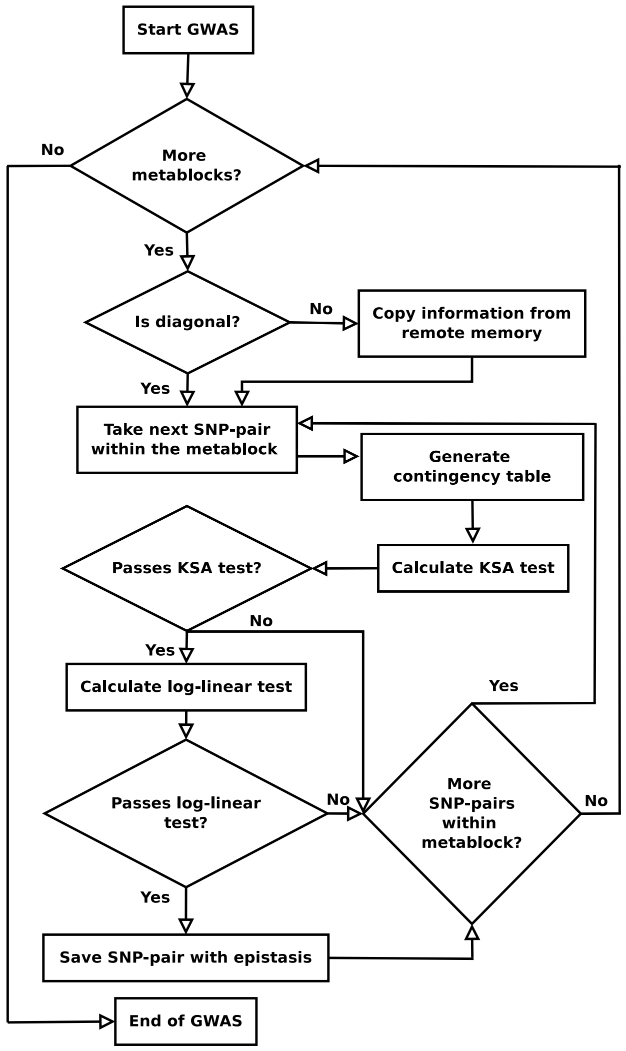
Start GWAS
More metablocks?
No
Yes
Is diagonal?
No
Copy information from remote memory
Yes
Take next SNP-pair within the metablock
Generate contingency table
Calculate KSA test
Passes KSA test?
No
Yes
Calculate log-linear test
Yes
Passes log-linear test?
No
More SNP-pairs within metablock?
No
Yes
Save SNP-pair with epistasis
End of GWAS

Figure 3. Procedure within each UPC++ process.

in Figure 2, Process 0 always computes metablocks that are in the first or fourth row/column). Note that this distribution perfectly balances the workload (same number of metablocks per process) if the number of processes is odd. Otherwise, half of the processes have to analyze one additional metablock. However, this unbalance is not significant for a large number of processes. **Other distributions are conceivable. In general, any employed distribution needs to fulfill the condition that each process computes metablocks where at least one of the blocks with biallelic information is allocated in its local memory.**

The flowchart in Figure 3 summarizes the behavior of each UPC++ process: for each associated metablock, the process initially gets one biallelic information block from remote memory if necessary (if the metablock is not diagonal) and then searches for epistasis in all the SNP-pairs within the metablock. Thanks to the shared global memory space and the one-sided communication available in the PGAS languages, the remote copies can be performed without synchronization with the owner. Note that the implementation is flexible enough to vary the statistical tests that determine if the SNP-pairs present epistatic interaction.

A. UPC++ Optimization Techniques

In order to increase the efficiency of the UPC++ parallelization, a set of optimization techniques have been applied, mostly focused on minimizing the communication cost:

- Data locality optimization. As previously explained, the workload distribution guarantees that, at least, one of the blocks with biallelic information used to compute each metablock does not need to be transferred from remote memory. ThereforeI think the paper could be made a little stronger with some more discussion of why UPC++ was chosen as a language. As far as I can tell no use of any features not in UPC was made, or maybe I am mistaken (?). Also, so discussion as to why a PGAS language and not MPI would be appreciated. What is it about the global address space that helps here ?, remote accesses are minimized and the overhead due to communication among processes reduced.
- Overlapping of computation and communication with asynchronous copies. Before starting to search for epistasis within a metablock, each process initializes the remote copies that will be necessary for the next metablock. As these copies are asynchronously performed, computation and communication can overlap.
- Aggregation of remote shared memory accesses. As modern networks usually move a large chunk of data more efficiently than smaller pieces, the processes get data from remote shared memory through bulk copies, using the `async_copy` function provided by UPC++ [11].
- Space privatization. As explained in Section III-B, global pointers save information about both the process ID and the local address. Consequently, updating the position of a global pointer requires updating both fields and thus it is slower than working with regular C++ pointers. When dealing with shared data with affinity to the local process the accesses are performed through regular C++ pointers after a cast from a global UPC++ pointer.

B. Pthreads Parallelization for Multi-Core Nodes

Increasing the number of UPC++ processes also increases the number of information blocks and metablocks and thus, more communication must be carried out. On multi-core clusters, instead of using one process per core, parallel programs can exploit the shared memory within the nodes for a multi-threaded computation. We provide a hybrid UPC++/Pthreads parallelization which is flexible enough to allow the users to choose the number of UPC++ processes and Pthreads per node. This hybrid approach has been successfully applied for parallel numerical routines merging UPC either with OpenMP [28] or Pthreads [33].

The hybrid implementation uses the same data distribution among UPC++ processes. The only difference is that each process generates T Pthreads to compute each metablock. Each Pthread calculates the contingency table and carries

Table II
CHARACTERISTICS OF THE COMPUTER PLATFORMS USED IN THE TESTS.

	Pluton	**Edison**
Processor	Intel Xeon E5-2660 *Sandy Bridge*	Intel Xeon E5-4603 *Ivy Bridge*
Clock rate	2.20 GHz	2.4 GHz
Number of nodes	16	5,576
Processors per node	2	2
Cores per node	16	24
L1 cache	32 KB (private)	32 KB (private)
L2 cache	256 KB (private)	256 KB (private)
L3 cache	20 MB (shared per processor)	30 MB (shared per processor)
Memory per node	64 GB	64 GB
Interconnect	InfiniBand FDR	Cray Aries

Table III
CHARACTERISTICS OF THE DATASETS USED IN THE TESTS.

	sim1	**sim2**	**sim3**	**WTCCCIbd**
Real data?	No	No	No	Yes
Number of SNPs	10,000	25,000	50,000	500,568
Number of cases	800	1,600	1,600	2,005
Number of controls	800	1,600	1,600	3,004
Seq. time *Pluton*	2 m	19 m	1 h 15 m	$>$ 6 days
Seq. time *Edison*	1 m	10 m	41 m	$>$ 3 days

out the KSA and, if necessary, the log linear tests for each SNP-pairs within the same metablock. Two SNP-pair distribution schemes within the metablocks have been implemented. Thanks to using the inheritance and polymorphism available in OOP languages, the distribution scheme can be selected during execution time through one parameter.

- Static distribution. The SNP-pairs are initially distributed among the Pthreads, such that each one searches for epistasis on the pairs associated to it. The current implementation uses the block distribution for the SNP-pairs. Nevertheless, block-cyclic and cyclic should obtain the same performance. The main drawback of this approach is the unbalanced workload: analyzing SNP-pairs that need to calculate the log-linear test is more expensive than analyzing pairs already discarded by the KSA filter. Therefore, those cores that process a higher percentage of SNP-pairs that do not pass the KSA test are idle while other threads still work.
- Dynamic distribution. The SNP-pairs are not initially associated to any Pthread. Each Pthread starts analyzing all the pairs that contain a certain SNP and, once they finish, they process the pairs for the next SNP that has not been studied yet. In this case the workload adapts to the complexity of each SNP-pair. However, we need a variable that records which is the next SNP to compute. Accesses to this variable must be synchronized with a semaphore or mutex, which might lead to runtime overhead.

As previously explained, the main advantage of using Pthreads for the intra-node computation is the reduction of the number of UPC++ processes and thus the amount of communication. Additionally, less UPC++ processes lead to less context switching overhead. Nevertheless, all cores associated to the Pthreads must be synchronized at the end of the metablock computation, which delays the program. The experimental evaluation in the next section will provide an insight to the scenarios where using Pthreads is recommendable and which distribution yields the best scalability.

V. EXPERIMENTAL EVALUATION

Two platforms, with different characteristics (see details in Table II), are used for evaluating the scalability of our UPC++ GWAS implementation and for analyzing the impact of the implementation decisions on performance:

- *Pluton*. This system, installed at the Computer Architecture Group at the University of A Coruña [34], consists of 16 nodes, each of them with 2 octa-core Intel Xeon E5-2660 processors (16 cores at 2.20 Ghz per node) and 64 GB of memory. The compute nodes are interconnected by a fast InfiniBand FDR network (56 Gbps of theoretical bandwidth).
- *Edison*. This supercomputer is a novel Cray XC30 system with 133,824 compute cores and 357 TB of total memory installed at the National Energy Research Scientific Computing Center (NERSC) [35]. Each Cray XC30 node has 24 cores, grouped by 12 in 2 processors. Internode communication is performed through the custom Cray Aries Network, which is a high-bandwidth and low-latency interconnect with Dragonfly topology and hardware Remote Direct Memory Access (RDMA) support.

Three datasets of biallelic information with different number of SNPs and individuals have been simulated in order to evaluate the performance. Additionally, we also used one real-world dataset obtained from the WTCCC project [36], which contains information with 2,005 cases of bipolar disorder disease and 3,004 controls. Table III summarizes the characteristics of the datasets, including the sequential time (using only one core) to perform the GWAS analysis with the KSA and log-linear tests on *Pluton* and *Edison*.

All the graphs in this section will use parallel efficiency as the measure to determine the performance of the application. It represents the percentage of the resources that are actually exploited during the parallel execution. Let C be the number of cores used in the experiment, T_{seq} and T_c the sequential and parallel time, respectively. Efficiency is defined as: $\frac{T_{seq}*100}{T_c*C}$

A. Best Approach for Intra-Node Parallelization

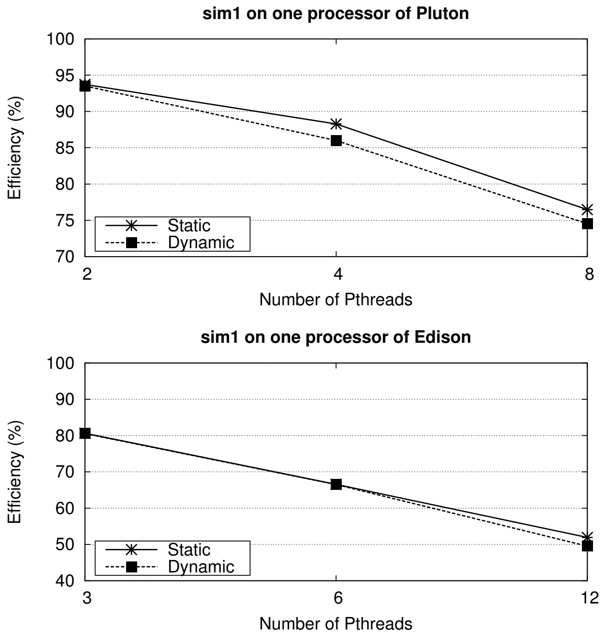

Figure 4. Evaluation of the data distributions schemes for the Pthreads intra-node parallelization.

Firstly, we evaluate which configuration for our application better exploits the internal resources of the nodes. The graphs in Figure 4 compare the performance of the two Pthreads distribution schemes explained in Section IV-B (static and dynamic) on both experimental platforms when using only one UPC++ process and, thus, parallelizing only with Pthreads. As only the cores within one process with shared memory are used (8 or 12 for *Pluton* or *Edison*, respectively), only the smallest dataset (*sim1*) is tested. The experimental results show that the static distribution yields slightly better performance. Although the workload might be unbalanced if some Pthreads analyze more SNP-pairs that pass the KSA filter, the impact of this unbalance on performance is less significant than the synchronizations needed in the dynamic distribution to keep trace of the not analyzed SNP-pairs.

However, the scalability is not high even for the static distribution as all Pthreads must be synchronized when each metablock is computed. Therefore, we have analyzed whether using Pthreads for intra-node computation is efficient. Several configurations with variable number of UPC++ processes, number of Pthreads per process, number of blocks per process and datasets have been executed for this purpose. The graphs of Figure 5 compare the efficiency of our parallel GWAS when varying the ratio of UPC++ processes/Pthreads for the *sim2* and *sim3* datasets. Each line represents a different number of Pthreads per UPC++ process using the static SNP-pair distribution scheme (the one with the highest efficiency). In order to use the same number of cores (C) for each line and provide a fair comparison, only C/T UPC++ processes are launched when using T Pthreads. For simplicity, the graphs only show the results for the best number of blocks per process on each scenario (the influence of this parameter will be studied in the next subsection) and using at least one whole node. For *Edison* only results up to a representative number of nodes (64) are obtained. This number of nodes is enough to understand the influence of the number of Pthreads on the efficiency. All the 256 cores are used on *Pluton*.

Three of the four studied scenarios obtain the best results by mapping one UPC++ process to each core, and thus avoiding the overhead of creating and synchronizing Pthreads. Furthermore, the efficiency of this UPC++-only approach is very high for these three scenarios: over 80% on *Pluton* for both datasets and on *Edison* with *sim3* for any number of cores, even though the execution times are sometimes quite short (e.g. only 2 seconds for 1536 cores).

Nevertheless, the comparison results on *Edison* with *sim2* depend on the number of cores. Up to 48 cores, using Pthreads is less efficient than UPC++-only. From 96 to 768 cores, a small number of Pthreads is more efficient. For 1,536 cores, the best performance is obtained with only one UPC++ process per processor, filling the cores with Pthreads. In these cases the hybrid UPC++/Pthreads approach obtains better efficiency because the execution times are short (note from Table III that sequential computation on *Edison* is faster than on *Pluton*). Consequently, the usage of many UPC++ process leads to many copies of small size, significantly increasing the communication overhead.

Therefore, we can conclude that the efficiency is improved by mixing UPC++ and Pthreads when the GWAS is executed on a strong scaling scenario (many cores and not very large datasets). Otherwise, one UPC++ process per core should be used.

B. Influence of the Number of Blocks per Process

Figure 6 illustrates the evolution of the efficiency depending on the number of blocks per process used in the distribution, which is a configurable parameter in our implementation. Only experimental results for the best ratio of UPC++ processes to Pthreads are shown for each scenario. Furthermore, only measurements for 1, 2 and 4 blocks per process are shown, since dividing the information in more than 4 blocks per process always decreases performance. The results indicate two trends. Firstly, the usage of only one block per process is more efficient for a large number of cores (more than 192 cores on *Edison*). For such a number of cores we must avoid generating more blocks that would be copied from all processes, increasing the communication overhead. Anyway, as expected, the influence of this overhead is less significant for larger datasets (*sim3*). Secondly, the division of the computation in 2

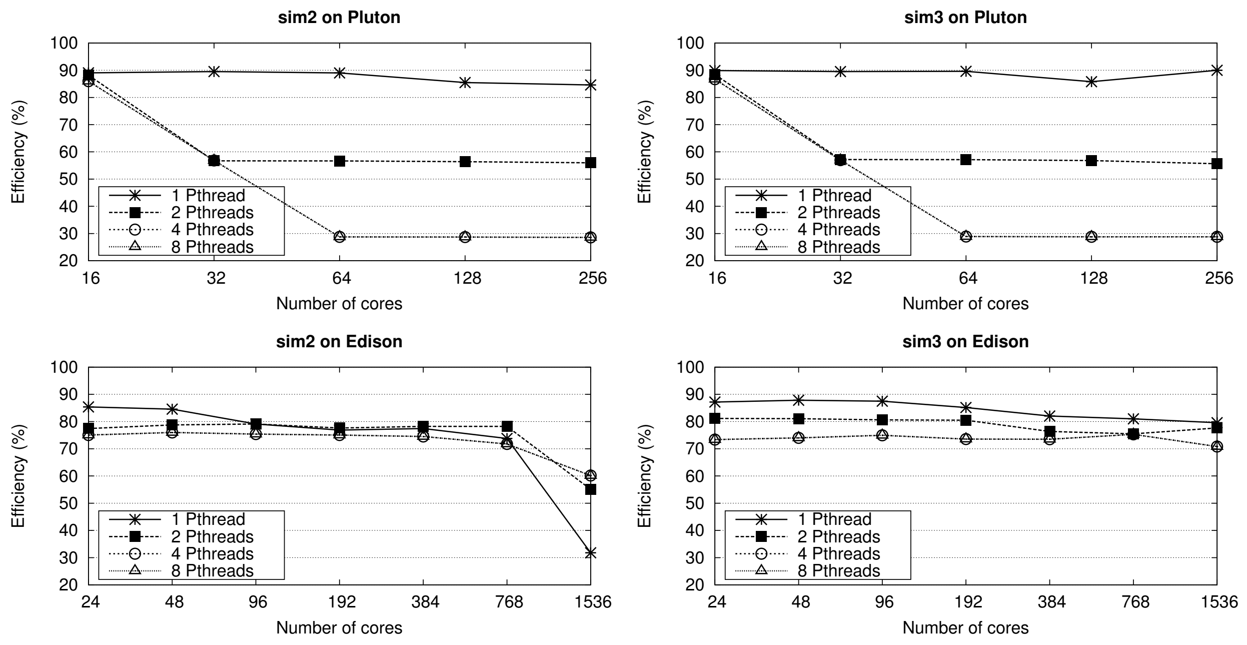

Figure 5. Efficiency of the parallel GWAS varying the number of Pthreads per UPC++ process.

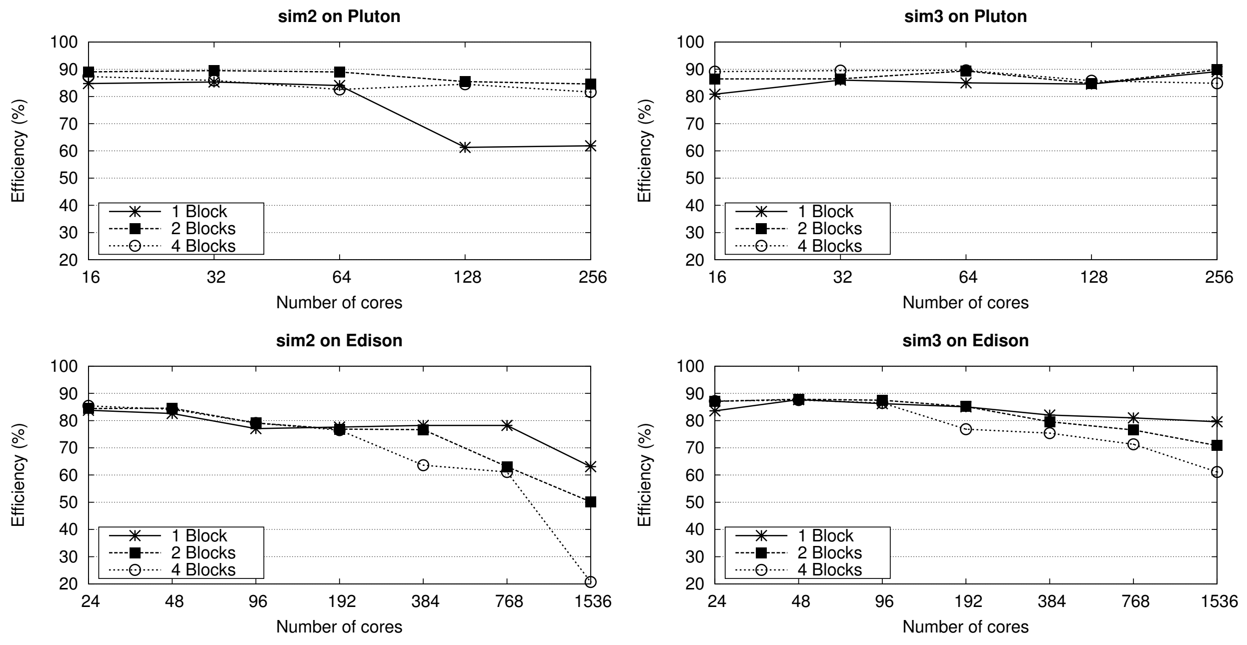

Figure 6. Efficiency of the parallel GWAS varying the number of blocks per UPC++ process.

or 4 blocks per process is more efficient for a moderate number of cores, although the improvement is not very significant.

C. *Comparison with Other Parallel Architectures*

Finally, we compare the execution time of the UPC++ GWAS implementation to GBOOST [7], a CUDA version that applies the same statistical filters in order to search for epistatic interactions (explained in Section III-A) on GPUs. Table IV, shows some representative results for the *WTCCCIbd* dataset, with about 500,000 SNPs from about 5,000 real samples. Note from Table III that even for this medium-size dataset the sequential GWAS analysis needs more than 6 and 3 days on *Pluton* and *Edison*, respectively.

Regarding the GPUs, a NVIDIA GTX 750Ti analyzes all the SNP-pairs in slightly more than 2.5 hours, while the high-end GTX Titan only needs about 1 hour. Nevertheless, the UPC++ version significantly outperforms these, even on the relatively small *Pluton* cluster. Executing on a large and

Table IV
PERFORMANCE COMPARISON ON DIFFERENT PARALLEL PLATFORMS WHEN LOOKING FOR EPISTASIS IN THE *WTCCCIbd* DATASET.

Platform	Time
Edison (12288 cores)	45 s
Edison (1536 cores)	5 m
Pluton (256 cores)	45 m
NVIDIA GTX Titan	1h 01 m
NVIDIA GTX 750Ti	2h 41 m

modern supercomputer like *Edison*, we do not need to ask for all the resources in order to analyze the dataset in a very short period of time: we need 5 minutes when using 1536 cores and less than one minute on 12,288 cores. This performance comparison has been carried out with the WTCCC dataset since it is the largest publicly available dataset to download. However, even a more significant difference between the UPC++ and the CUDA execution time can be expected for larger-scale datasets (typically consisting of around 5M SNPs and 100,000 individuals). For those datasets we could use additional resources on *Edison* or other large supercomputers to run the UPC++ version in only a few minutes.

VI. CONCLUSIONS

Recent advances in high-throughput genotyping technologies establish the need for fast implementations of statistical epistasis in GWAS. Recent work has shown how GPUs and FPGAs can be used to speed up such methods. In this article we provide an implementation for multi-core clusters and we demonstrate that these studies can be accelerated using this type of platforms. The parallel GWAS has been implemented using UPC++, a new PGAS extension of C++, in order to take advantage of the strengths of the PGAS model (e.g. one-sided communication or direct access to remote data) and the OOP paradigm (e.g. inheritance or polymorphism).

Therefore, the novelty of this work is two-fold. On the one hand, this is the first UPC++ implementation of a bioinformatics algorithm (and, in general, of any big data problem). Thanks to the optimization techniques included in the code **and to the use of efficient one-sided communications available in PGAS languages, the communication overhead can be significantly reduced.** Consequently, the parallel efficiency is higher than 80% to study 50,000 SNPs from 3,200 individuals, when using 1,536 cores of a Cray XC30 supercomputer (only 2 seconds of execution time). Our approach also includes the option to launch Pthreads on each UPC++ process to improve performance on strong scaling scenarios. On the other hand, **we provide a parallel GWAS version based on regression models** that can exploit the hardware capabilities of clusters and supercomputers. Our evaluation shows that 12,288 cores of the mentioned Cray XC30 supercomputer only need 45 seconds to analyze about 500,000 SNPs from about 5,000 real samples, while one of the most powerful available GPUs needs around one hour using the same statistical tests.

ACKNOWLEDGEMENTS

This study makes use of data generated by the Wellcome Trust Case-Control Consortium. A full list of the investigators who contributed to the generation of the data is available from www.wtccc.org.uk. Funding for the project was provided by the Wellcome Trust under award 076113 and 085475. We gratefully thank the Computer Architecture Group at the University of A Coruña and the National Energy Research Scientific Computing Center for providing access to the *Pluton* and *Edison* clusters, respectively. We also thank Dr. Yili Zheng for his help to install the UPC++ compiler on these systems.

REFERENCES

[1] B. Maher, "Personal Genomes: the Case of the Missing Heritability," *Nature*, vol. 456, no. 7218, pp. 18–21, 2008.
[2] P. C. Phillips, "Epistasis, the Essential Role of Gene Interactions in the Structure and Evolution of Genetic Systems," *Nature Review Genetics*, vol. 9, no. 11, pp. 855–867, 2008.
[3] J. H. Moore, F. W. Asselbergs, and S. M. Williams, "Bioinformatics Challenges for Genome-Wide Association Studies," *Bioinformatics*, vol. 26, no. 4, pp. 445–455, 2010.
[4] H. J. Cordell, "Detecting Gene-Gene Interactions that Underlie Human Diseases," *Nature Review Genetics*, vol. 10, no. 6, pp. 392–404, 2009.
[5] K. van Steen, "Travelling the World of Gene–Gene Interactions," *Briefings in Bioinformatics*, vol. 13, no. 1, pp. 1–19, 2011.
[6] Y. Wang, G. Liu, M. Feng, and L. Wong, "An Empirical Comparison of Several Recent Epistatic Interaction Detection Methods," *Bioinformatics*, vol. 27, no. 21, pp. 2936–2943, 2011.
[7] L. S. Yung, C. Yang, X. Wan, and W. Yu, "GBOOST: A GPU-Based Tool for Detecting Gene-Gene Interactions in Genome-Wide Case Control Studies," *Bioinformatics*, vol. 27, no. 9, pp. 1309–1310, 2011.
[8] B. Goudey, D. Rawlinson, Q. Wang, and et al, "GWIS - Model-Free, Fast and Exhaustive Search for Epistatic Interactions in Case-Control GWAS," *BMC Genomics*, vol. 14, no. Supl 3, 2012.
[9] J. Piriyapongsa, C. Ngamphiw, A. Intarapanich, S. Kulawonganunchai, A. Assawamakin, C. Bootchai, P. J. Shaw, and S. Tongsima1, "iLOCi: a SNP Interaction Priorization Technique for Detecting Epistasis in Genome-Wide Association Studies," *BMC Genomics*, vol. 13, no. Supl 7, 2012.
[10] L. Wienbrandt, J. C. Kässens, J. González-Domínguez, B. Schmidt, D. Ellinghaus, and M. Schimmler, "FPGA-Based Acceleration of Detecting Statistical Epistasis in GWAS," in *Proc. 14th Intl. Conf. on Computational Science (ICCS'14)*, Cairns, Australia, 2014.
[11] Y. Zheng, A. Kamil, M. Driscoll, H. Shan, and K. Yelick, "UPC++: a PGAS Extension for C++," in *Proc. 28th IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS'14)*, Phoenix, AR, USA, 2014.
[12] J. Zhao and L. Jin, "Test for Interaction Between Two Unlinked Loci," *The American Journal of Human Genetics*, vol. 78, no. 1, pp. 15–27, 2006.
[13] S. Purcell, B. Neale, T.-B. K, and et al, "PLINK: a Tool Set for Whole-Genome Association and Population-Based Linkage Analyses," *The American Journal of Human Genetics*, vol. 81, no. 3, pp. 559–575, 2007.
[14] A. Gyenesei, J. Moody, C. A. Semple, C. S. Haley, and W. Wei, "High-Throughput Analysis of Epistasis in Genome-Wide Association Studies with BiForce," *Bioinformatics*, vol. 28, no. 15, pp. 1957–1964, 2012.
[15] C. Yang, Z. He, X. Wan, Q. Yang, H. Xue, and W. Yu, "SNPHarvester: a Filtering-Based Approach for Detecting Epistatic Interaction in Genome-Wide Association Studies," *Bioinformatics*, vol. 25, no. 4, pp. 504–511, 2009.
[16] X. Wan, C. Yang, Q. Yang, H. Xue, N. L. Tang, and W. Yu, "Predictive Rule Inference for Epistatic Interaction Detection in Genome-Wide Association Studies," *Bioinformatics*, vol. 26, no. 1, pp. 30–37, 2010.
[17] S. Prabhu and I. Peer, "Ultrafast Genome-Wide Scan for SNP-SNP Interactions in Common Complex Disease," *Genome Research*, vol. 22, no. 11, pp. 2230–2240, 2012.
[18] X. Wan, C. Yang, Q. Yang, H. Xue, X. Fan, N. L. Tang, and W. Yu, "BOOST: a Fast Approach to Detecting Gene-Gene Interactions in Genome-Wide Case-Control Studies," *The American Journal of Human Genetics*, vol. 87, no. 3, pp. 325–340, 2010.

[19] R. L. Milne, J. Herranz, K. Michailidou, and et al, "A Large-Scale Assessment of Two-Way SNP Interactions in Breast Cancer Susceptibility Using 46,450 Cases and 42,461 Controls from the Breast Cancer Association Consortium," *Human Molecular Genetics*, vol. 23, no. 7, pp. 1934–1946, 2014.

[20] M. Chu, R. Zhang, Y. Zhao, and et al, "A Genome-Wide Gene-Gene Interaction Analysis Identifies an Epistatic Gene Pair for Lung Cancer Susceptibility in Han Chinese," *Cancinogenesis*, vol. 32, no. 3, pp. 572–577, 2014.

[21] J. Bi, J. Gelernter, J. Sun, and H. R. Kranzler, "Comparing the Utility of Homogeneous Subtypes of Cocaine Use and Related Behaviors with DSM-IV Cocaine Dependence as Traits for Genetic Association Analysis," *American Journal of Medical Genetics*, vol. 165, no. 2, pp. 148–156, 2014.

[22] L. Ma, H. B. Runesha, D. Dvorkin, J. R. Garbe, and Y. Da, "Parallel and Serial Computing Tools for Testing Single-Locus and Epistatic SNP Effects of Quantitative Traits in Genome-Wide Association Studies," *BMC Bioinformatics*, vol. 9, no. 315, 2008.

[23] M. A. Steffens, T. A. Becker, T. Sander, and et al, "Feasible and Successful: Genome-Wide Interaction Analysis Involving All 1.9x1011 Pair-Wise Interaction Tests," *Human Heredity*, vol. 69, no. 4, pp. 268–284, 2010.

[24] UPC Consortium, "UPC Language Specifications, v1.2," http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf.

[25] T. El-Ghazawi, W. Carlson, T. L. Sterling, and K. Yelick, *UPC: Distributed Shared-Memory Programming*. John Wiley & Sons Inc, 2005.

[26] R. W. Numrich and J. Reid, "Co-Array Fortran for Parallel Programming," *ACM FORTRAN FORUM*, vol. 17, no. 2, pp. 1–31, 1998.

[27] K. Yelick, L. Semenzato, G. Pike, and et al, " Titanium: A High-Performance Java Dialect," *Concurrency: Practice and Experience*, vol. 10, no. 11, pp. 825–836, 1998.

[28] J. González-Domínguez, M. J. Martín, G. L. Taboada, J. Touriño, R. Doallo, D. A. Mallón, and B. Wibecan, "UPCBLAS: a Library for Parallel Matrix Computations in Unified Parallel C," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 14, pp. 1645–1667, 2012.

[29] J. González-Domínguez, O. A. Marques, M. J. Martín, G. L. Taboada, and J. Touriño, "Design and Performance Issues of Cholesky and LU Solvers Using UPCBLAS," in *Proc. 10th IEEE Intl. Symp. on Parallel and Distributed Processing with Applications (ISPA'12)*, Leganés, Spain, 2012, pp. 40–47.

[30] C. Bell, D. Bonachaea, R. Nishtala, and K. Yelick, "Optimizing Bandwidth Limited Problems Using One-Sided Communication and Overlap," in *Proc. 20th IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS'06)*, Rhodes Island, Greece, 2006.

[31] R. Nishtala, P. Hargrove, D. Bonachea, and K. Yelick, "Scaling Communication-Intensive Applications on BlueGene/P Using One-Sided Communication and Overlap," in *Proc. 23rd IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS'09)*, Rome, Italy, 2009.

[32] R. Nishtala, Y. Zheng, P. Hargrove, and K. Yelick, "Tuning Collective Communication for Partitioned Global Address Space Programming Models," *Parallel Computing*, vol. 37, no. 9, pp. 576–591, 2011.

[33] E. Georganas, J. González-Domínguez, E. Solomonik, Y. Zheng, J. Touriño, and K. Yelick, "Communication Avoiding and Overlapping for Numerical Linear Algebra," in *Proc. 24th ACM/IEEE Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC'12)*, Salt Lake City, UT, USA, 2012.

[34] Computer Architectures Group. University of A Coruña, http://www.gac.udc.es.

[35] National Energy Research Scientific Computing Center (NERSC), http://www.nersc.gov.

[36] The Wellcome Trust Case Control Consortium, "Genome-wide association study of 14,000 cases of seven common diseases and 3,000 shared controls," *Nature*, vol. 447, no. 7145, pp. 661–78, 2007.