A Simple Multi-Core Parallelization Strategy for Learning Classifier System Evaluation

James Rudd
Dartmouth College
1 Medical Center Dr.
Lebanon, NH 03755,USA
james.e.rudd.gr@dartmouth.edu

Jason H. Moore
Dartmouth College
1 Medical Center Dr.
Lebanon, NH 03755,USA
jason.h.moore@dartmouth.edu

Ryan J. Urbanowicz
Dartmouth College
1 Medical Center Dr.
Lebanon, NH 03755,USA
ryan.j.urbanowicz@dartmouth.edu

ABSTRACT

Permutation strategies for statistically evaluating the significance of predictions and patterns identified within learning classifier systems (LCSs) have only appeared since 2012. While already considered to be computationally expensive algorithms, a permutation testing based approach to determining statistical significance has the potential to be many times more demanding. One area of LCS research which has become both feasible and popularized in recent years is the adoption of parallelization strategies. In the present study we explore the simple benefits of parallelizing a set of LCS analyses in an attempt to make the completion of a permutation test with cross validation more feasible on a single multi-core workstation. We test our python implementation of this strategy in the context of a simulated complex genetic epidemiological data mining problem. Our evaluations indicate that on Windows 7 computers, as long as the number of concurrent processes does not exceed the number of CPU cores, the speedup achieved is approximately linear.

Categories and Subject Descriptors

J.3 [**Computer Applications**]: Life and Medical Sciences—*biology and genetics*

General Terms

Algorithms, Performance, Design

Keywords

LCS, GBML, AF-UCS, parallelization, scalability, multi-core processors

GECCO'13 Companion, July 6–10, 2013, Amsterdam, The Netherlands.

1. INTRODUCTION

Large scale investigations of genetic variation related to human disease have become increasingly complicated by the acknowledgement of, and search for complex patterns of association, including multivariate effects, epistatic interactions, and heterogeneous relationships [12]. Previously, we introduced a promising new methodology to address these complexities using a Learning Classifier System (LCS) algorithm [14]. Learning classifier systems (LCSs) [18] are a rule-based class of algorithms which combine machine learning with evolutionary computing and other heuristics to produce an adaptive system. LCSs represent solutions as sets of rules affording them the ability to learn iteratively, form niches, and adapt. These characteristics make the application of LCSs to the problem of heterogeneity, in particular, intrinsically appealing.

In [14] we applied our own extended supervised-learning classifier system, called AF-UCS [13], and a statistical and visualization guided knowledge discovery pipeline [15] to a real world genetic epidemiology study of bladder cancer susceptibility. In that work, we successfully replicated the identification of previously characterized factors that modify bladder cancer risk: i.e. single nucleotide polymorphisms from a DNA repair gene, and smoking. Furthermore, we identified potentially heterogeneous groups of subjects characterized by distinct patterns of association. While successful, this study was performed on a relatively small dataset, and the computation was aided by a 1576 processor cluster, a resource to which few researchers may have access. Due to the complexity of LCS and the demands of large-scale data mining, the issue of scalability remains both a key challenge and opportunity for the LCS research community [5].

In order to obtain statistical significance measures in [14], k-fold cross validation (CV) was paired with p-fold permutation testing (where $k = 10$ and $p = 1000$). While CV had been previously applied in various LCS studies, it had yet to be combined with permutation testing for LCS significance evaluations. CV has typically been utilized to determine average testing accuracy and account for algorithm over-fitting. CV is performed by randomly partitioning a dataset into k equal partitions and applying the algorithm k separate times during which $k - 1$ partitions are used to train the algorithm, and the remaining partition is set aside for testing the resulting model. Permutation testing offers a non-parametric strategy for evaluating whether an observed test statistic (such as test accuracy) is significantly different from what might be observed by random chance. This

characterization as a non-parametric strategy is particularly important in LCS evaluations where the probability distribution of different statistics of interest would not be known ahead of time. This is critical to LCS data mining, in that it offers researchers a measure of confidence when evaluating algorithm performance or extracting knowledge from the rule population.

Permutation testing yields a null distribution for a given target statistic by repeating the analysis on variations of the dataset (with class status shuffled). This null distribution is then used to determine the likelihood that the observed result could have occurred by chance. In [14], 1000 permuted versions of the original dataset were generated by randomly permuting the affection status (class) of all samples, while preserving the number of cases and controls. It should be noted that for each permuted dataset the algorithm was run using 10-fold CV. Thus, this testing strategy required $k * p$ or 10,000 runs of the algorithm in total. Without access to large scale multi-processor clusters (i.e. completed serially on a single computer workstation), this task quickly becomes impractical.

Parallelization presents one strategy to ameliorate the cost of running LCS repeatedly for both CV and permutation testing. The time complexity of LCS algorithms, specifically those of the Michigan style, are generally bounded by the number of generations used to evolve the solution set. Due to the inherent data dependency between each iteration of rule set generations, parallelization of this major term in the asymptotic time analysis is not feasible. Previous works have focused on parallelizing mechanisms of the LCS algorithm itself using General Purpose Graphics Processing Units (GPGPUs) with NVIDIA's Compute Unified Device Architecure (CUDA). These included strategies to parallelize (1) matching in XCS [10], (2) fitness calculation in BioHEL, and (3) prediction computation (also in XCS) [11]. While these strategies successfully decrease the time burden of LCS, gains may also be achieved through careful consideration of the analytical workflow. Specifically, since both cross validation and permutation testing are "embarrassingly parallel", there is a clear opportunity for performance improvement through running the individual instances of the LCS algorithm concurrently.

In the present study, we have implemented a modified version of AF-UCS which capitalizes on the multi-core architecture of most modern computers. Consistent with parallelization work in other python projects [9, 8, 7], we use the multiprocessing [3] module in Python 2.6 and greater to enable AF-UCS to launch multiple instances concurrently. This enables AF-UCS to internally manage both CV and permutations parallelized over processes run on separate cores of the CPU. Further, we show that use of this implementation on typical windows desktops can offer significant time savings without the use of enthusiast level hardware. The remainder of this paper is organized as follows. Methods for the implementation and evaluation of this strategy are given in Section 2. The results with discussion are given in section 3. Conclusions are drawn and ongoing efforts outlined in section 4.

2. METHODS

In this section we describe (1) the LCS algorithm and the run parameters used, (2) implementation of the parallelization, (3) the evaluation strategy and benchmark dataset, and (4) the hardware utilized in testing.

2.1 AF-UCS

In order to implement and test our parallelization scheme, we used the Python encoding of AF-UCS, described in [13]. AF-UCS (attribute feedback UCS), is an expanded and modified implementation of UCS [6]. UCS, or the sUpervised Classifier System, is a michigan style LCS based largely on the popular XCS algorithm [20] but replaced reinforcement learning with supervised learning. UCS was designed specifically to address single-step problems such as classification and data mining, where delayed reward is irrelevant, and showed particular promise when applied to epistasis and heterogeneity in [19].

The selection of AF-UCS run parameters for this evaluation was arbitrary. We adopted mostly default michigan-style LCS run parameters. Parameters unique to this study include: 10,000 learning iterations, a rule population size of 1000, tournament selection, uniform crossover, and subsumption on. Parallelization code was incorporated into the Main.py class. The implementation described above is available on request (*ryanurbanowicz@gmail.com*) and will be posted on the LCS and GBML Central webpage [1].

2.2 Implementation

While multithreading presents an ideal implementation strategy for this problem as it would limit the overhead associated with new jobs and allow each new AF-UCS run to share a common state and data, the use of threads in Python suffers from the Global Interpreter Lock (GIL). The GIL is a well known feature of Python which allows only one thread to execute at a time. To circumvent this, we chose a multiprocessing strategy. Our parallelization of AF-UCS hinges on the 'Pool' object provided by the 'multiprocessing' package in Python (v 2.6 and greater) [3]. As shown in Algorithm 1, the size of the pool is initialized at the start based on a user modifiable input parameter which specifies the number of processes ($nProcesses$). Next, the input data is partitioned into k partitions in preparation for CV. Each CV job is then submitted to the process pool. For each of the ($nPermutations$) permutations specified by the user, the data labels are scrambled then submitted to the process pool. Finally, the jobs in the process pool are executed $nProcesses$ at a time. Thus, the number of jobs submitted to the pool depends both on the number of CV partitions and the number of permutations. While the permutation threads are relatively independent of each other, the CV processes do require shared input data which must be copied to each process and a final synchronization in order to calculate the average testing accuracy. To improve performance, we delay the CV synchronization until the end of the run at which time we also perform a permutation synchronization in order to generate a p-value based on the null distribution.

2.3 Evaluation

Test runs were submitted, and wall-times to the millisecond were recorded using the 'Measure-Command' provided by Windows Powershell [4]. Evaluations were completed on three separate workstations and all tests were completed after a fresh boot of the respective workstation. The hardware specifications for each machine are given in the next section. On each machine we ran three separate LCS analysis scenar-

Table 1: Hardware Specifications

Desktop 1		Desktop 2		Laptop	
Part	**Description**	**Part**	**Description**	**Part**	**Description**
CPU	Intel Core i7 950	**CPU**	Intel Xeon E3-1225	**CPU**	Intel Core i7-3840QM
RAM	12GB DDR3 1600Mhz	**RAM**	8GB DDR3 667 MHz	**RAM**	16 GB DDR3 798 Mhz
Hard Drive	Western Digital 7200 RPM	**Hard Drive**	Samsung PM830 238GB SSD	**Hard Drive**	Samsung PM830 477 GB SSD
	1TB SATA 3Gb/s		SATA Gen 3.0 6Gb/s		SATA Gen 3.0 6Gb/s
Cores	4	**Cores**	4	**Cores**	4
Processes	8	**Processes**	4	**Processes**	8
OS	Windows 7 x64	**OS**	Windows 7 x64	**OS**	Windows 7 x64
Python	2.7	**Python**	2.7	**Python**	2.7

Algorithm 1 Pseudo Code for CPU Parallelization

Require: $data, nGen, k, nPermutations, nProcesses$
  $jobPool = Pool(nProcesses)$
  $partitions = Partition(data, k)$
  for $part \in partitions$ **do**
    $jobPool.add(LCS(part, nGen))$
  end for
  for $permute \in nPermutations$ **do**
    $data = Permute(data)$
    $partitions = Partition(data, k)$
    for $part \in nPartitions$ **do**
      $jobPool.add(LCS(part, nGen))$
    end for
  end for
  $result = jobPool.execute()$
  $accuracy = result.averageTestAccuracy()$
  $sig = result.significance(alpha = .05)$
  return $result, accuracy, sig$

ios: (1) 10-fold CV alone, (2) 100-fold permutation testing alone, and (3) both 10-fold CV and 100-fold permutation testing. In order to decrease the amount of time necessary for these analyses, we opted to examine only 100-fold permutation testing. We expect that the wall-time results can be extrapolated to 1000-fold permutation testing in a linear fashion. Though the wall-time results are informative, we focus on the speedup and efficiency of our implementation in order to give an indication of expected performance on a variety of computer hardware.

Speedup and efficiency, both of which are computed from the raw wall-time data, are measures of the scalability of parallel implementations. Speedup for a specific number of processes indicates the fold improvement in wall-time relative to a single process serial implementation. Speedup with t processes is calculated as $S_t = \frac{Walltime_1}{Walltime_t}$, where $Walltime_1$ is the total wall-time for the analysis using only a single process. Efficiency is the slope of the speedup line for a specific number of processes. The efficiency of using t processes is simply calculated as $E_t = \frac{S_t}{t}$. Ideally, the wall-time fold change is equal to the total number of processes used which would result in a linear speedup and an efficiency value of 1.

In this evaluation we used a single simulated dataset as a benchmark for comparing across workstations, number of threads, and analysis scenarios. In keeping with our biological problem of interest, our benchmark dataset was simulated to concurrently possess patterns of epistasis and heterogeneity. The genetic models used to simulate our benchmark dataset were generated using GAMETES [17]. The datasets generated from these models were merged to produce our benchmark dataset containing two distinct underlying two-locus epistatic models, adding a heterogeneous component to the dataset. The first model was used to generate 75% of the dataset with a heritability of 0.05 and minor allele frequencies of 0.2. The second model was used to generate the remaining 25% of the dataset with a heritability of 0.025 and minor allele frequencies of 0.4. Both simulated models were selected to be of high difficulty based on model architecture according the model difficulty score prediction implemented in GAMETES [16]. This benchmark dataset included 200 instances, and a total of 20 attributes (4 of which were predictive). This dataset was selected to be extremely challenging such that AF-UCS would not be able to quickly converge on a solution.

2.4 Hardware

Three consumer computers were used to perform testing. The hardware details of these workstations are listed in Table 1. A range of multicore CPUs were used in order to test scaling and performance on hyperthreaded hardware.

3. RESULTS AND DISCUSSION

As shown in Figure 1, our parallelized algorithm scales approximately linearly up to the number of cores in representative consumer level computers. Workstation 1, Workstation 2, and the Laptop have quad-core processors and the speedup achieved using 1 to 4 processes is approximately linear. The Intel Core i7 processors (Workstation 1 and Laptop) make use of Intel Hyperthreading technology which allows two processes to execute concurrently using one CPU core. Thus, these CPUs offer eight concurrent processes of execution to the Operating System. Scaling results for 5 to 8 threads, however, is less than ideal. Generally, the hyperthreaded performance plateaus and little performance improvement is achieved. This is consistent across all three analysis scenarios with the exception of the CV scenario in which performance deteriorates with the addition of hyperthreaded processes. While Workstation 1 eventually achieves a 4-fold increase in performance in all 3 analyses, the Laptop achieves a maximum of approximately 3.5-fold. These results are consistent across CV, permutation testing, and CV combined with permutation testing. However, the CV and permutation testing analysis shows the most linear scaling for all three computers. This was the longest running analysis and the improved scaling suggest that process ini-

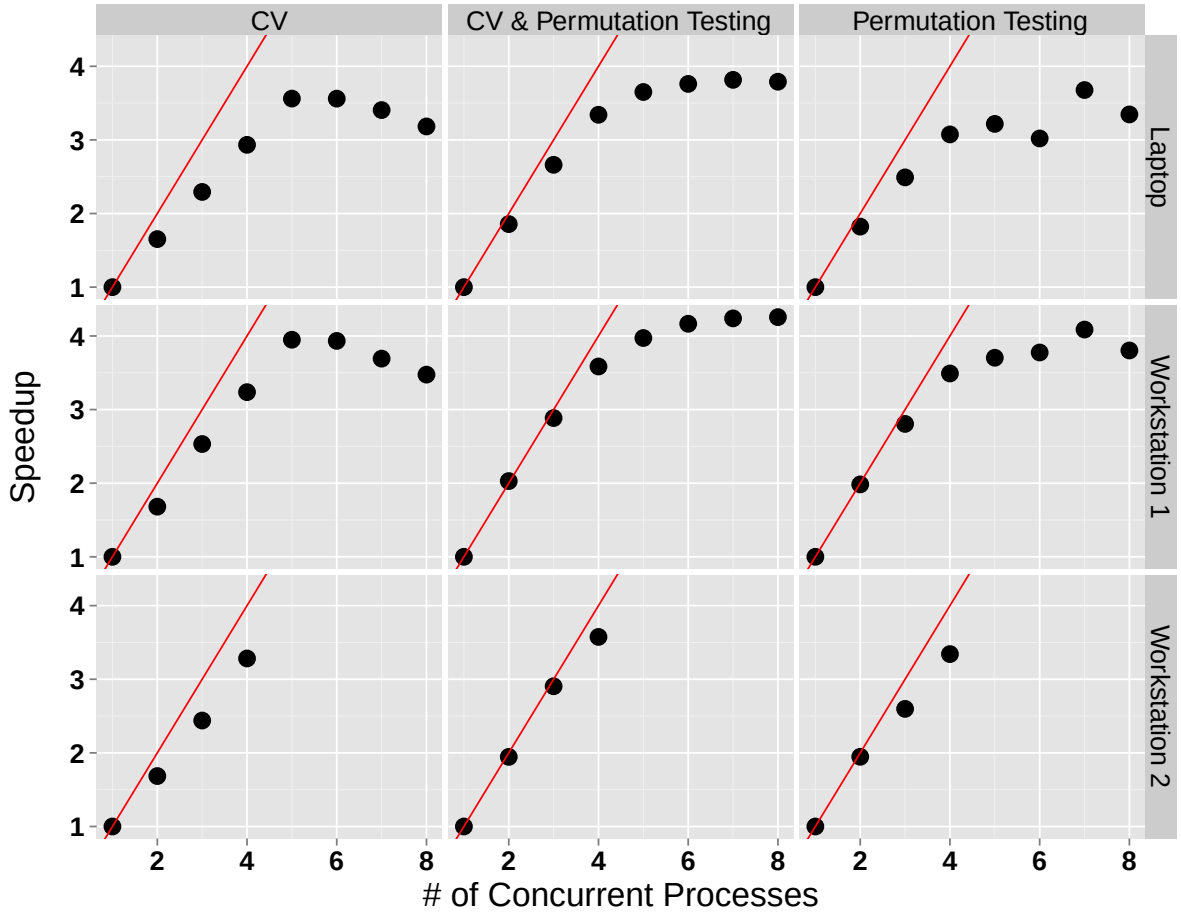

Figure 1: Speedup of parallelized AF-UCS by number of processes. Red lines indicate ideal linear scaling. Performance is approximately linear up to threads equal to the number of physical CPU cores.

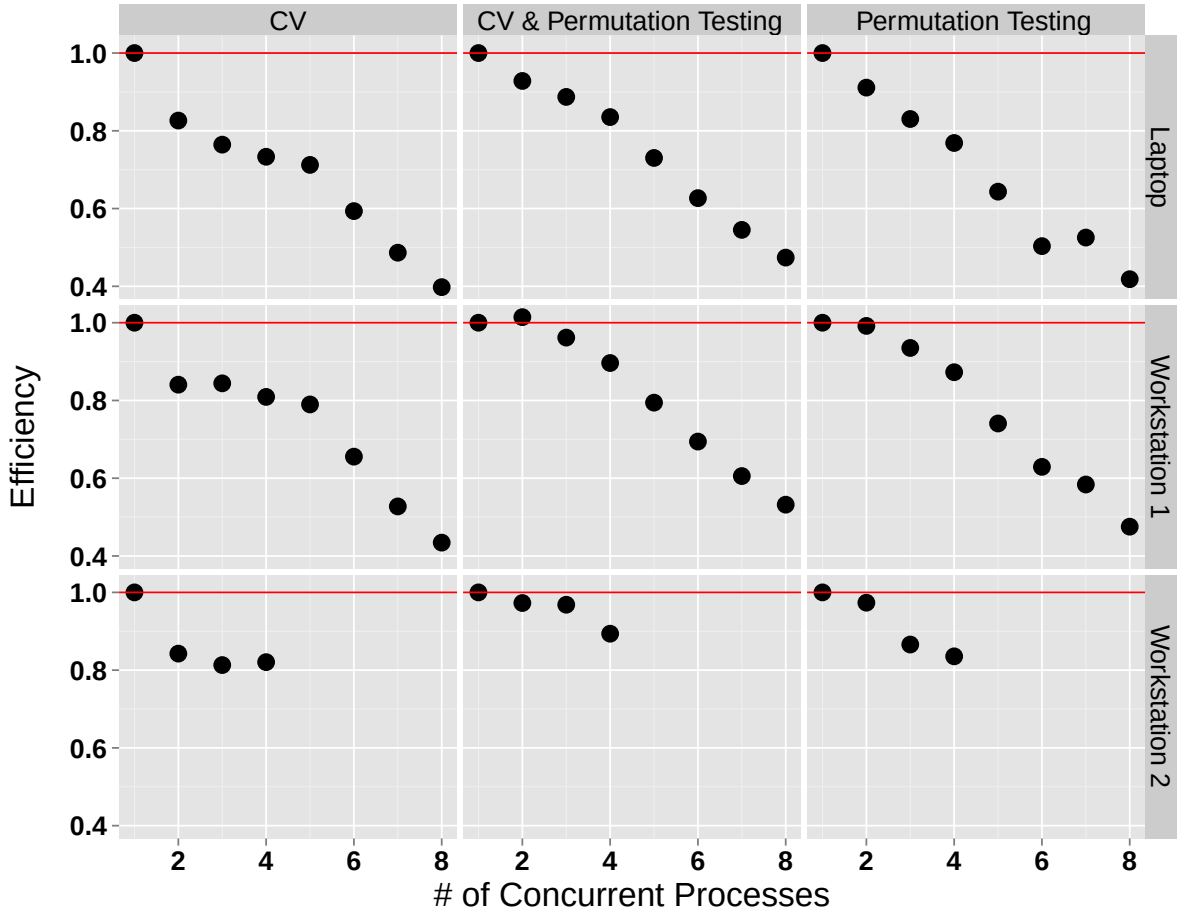

Figure 2: Efficiency of parallelized AF-UCS by number of processes. Red lines indicate ideal efficiency. Efficiency exceeds 80% up to threds equalling the number of physical CUP cores.

tialization costs were overshadowed by the LCS computation.

Figure 2 illustrates the efficiency of analysis over the different experimental run scenarios. Efficiency is the slope of the speedup curve which is ideally ≥ 1. The more the efficiency falls below 1, the less the additional thread contributes to overall performance increases. The results in Figure 2 are consistent with those in Figure 1, i.e. processes 1-4 achieve approximately 80% efficiency. However, as the number of processes exceeds the number of cores and the algorithm begins to make use of hyperthreading, efficiency drops precipitously.

While measures of scaling suffer when hyperthreading is used, there remain tangible benefits to making use of all concurrent processes. For example, using all 8 processes of Workstation 1 in the CV and permutation testing analysis yielded an approximate 1.2-fold increase in performance in comparison to using 4 processes (the number of physical CPU cores available). Relative to a single process, using all 8 processes increased performance by approximately 4.3-fold. In our small testing example, the improvement decreased the running time by approximately 1,672 and 29,100 seconds respectively. In larger analyses, however, the difference is potentially much greater. These data suggest a reliable operational heuristic of number of processes equal to the total number of concurrent processes available (including hyperthreading).

4. CONCLUSIONS

This study extends previous work by Urbanowicz *et. al.* [15, 14] by parallelizing AF-UCS in order to accelerate k-fold CV and permutation testing. While GPGPU based parallelization strategies can yield dramatic reductions in the run time of an algorithm, we focus on a CPU parallelization strategy that is likely to benefit a larger group of potential users in reducing the run time involved in performing permutation testing based statistical analysis in LCS algorithms. Our results show a consistent improvement in run time for CV, permutation testing, and CV combined with permutation testing, when parallelizing the analysis over the available cores of the CPU. As long as the number of processes does not exceed the number of CPU cores, the speedup achieved is approximately linear. This suggests a significant increase in performance for multi-core workstations which, we hope, will make this algorithm more approachable to a wider range of users. While a workstation with additional cores was not available for this study, the results suggest that additional CPU cores would likely yield similar, near-linear speedups. In the present study we have focused our implementation and evaluation on Windows 7 workstations. While this workstation selection was made largely on our own hardware availability we expect Windows 7 workstations to be as or more prevalent than Mac or Linux workstations in the research setting based on global market share statistics [2].

Additionally, it is worth noting that preliminary tests suggest that this implementation also works on Linux. We expect scaling and performance on Linux to be better than that of Windows 7 due to the multiprocessing module's use of the POSIX fork function on Linux. This fork allows new processes in Linux to be created with exact copies of the memory in the parent process. This in turn can decrease the initialization cost for each process.

This work constitutes a first step in the direction of adapting our AF-UCS algorithm to the computational demands inherent both in the determination of statistical significance as well as the analysis of large-scale data which are rapidly becoming more massive. Our future efforts will concentrate on further improvement of AF-UCS scalability as well as migration of the code base to an MPI based cluster implementation.

5. REFERENCES

[1] Genetics based machine learning central. *http://gbml.org/*.

[2] Netmarketshare: Market share statistics for internet technologies. *http://www.netmarketshare.com/*.

[3] Python multiprocessing module. *http://docs.python.org/2/library/multiprocessing.html*.

[4] Microsoft powershell. *http://technet.microsoft.com/en-us/library/bb978526.aspx*, 2012.

[5] J. Bacardit and X. Llorà. Large-scale data mining using genetics-based machine learning. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 3(1):37–61, 2013.

[6] E. Bernadó-Mansilla and J. M. Garrell-Guiu. Accuracy-based learning classifier systems: models, analysis and applications to classification tasks. *Evolutionary Computation*, 11(3):209–238, 2003.

[7] S. Binet, P. Calafiura, S. Snyder, W. Wiedenmann, and F. Winklmeier. Harnessing multicores: Strategies and implementations in atlas. In *Journal of Physics: Conference Series*, volume 219, page 042002. IOP Publishing, 2010.

[8] S. S. Foley, W. R. Elwasif, and D. E. Bernholdt. The integrated plasma simulator: A flexible python framework for coupled multiphysics simulation. *PyHPC 2011: Python for High Performance and Scientific Computing*, 2011.

[9] R. M. Friborg, J. M. Bjørndalen, and B. Vinter. Three unique implementations of processes for pycsp. *Communicating process architectures*, 2009:277–292, 2009.

[10] P. L. Lanzi and D. Loiacono. Speeding up matching in learning classifier systems using cuda. In *Learning Classifier Systems*, pages 1–20. Springer, 2010.

[11] D. Loiacono. Fast prediction computation in learning classifier systems using cuda. In *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, pages 169–170. ACM, 2011.

[12] J. H. Moore, F. W. Asselbergs, and S. M. Williams. Bioinformatics challenges for genome-wide association studies. *Bioinformatics*, 26(4):445–455, 2010.

[13] R. Urbanowicz, A. Granizo-Mackenzie, and J. Moore. Instance-linked attribute tracking and feedback for michigan-style supervised learning classifier systems. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, pages 927–934. ACM, 2012.

[14] R. J. Urbanowicz, A. S. Andrew, M. R. Karagas, and J. H. Moore. Role of genetic heterogeneity and epistasis in bladder cancer susceptibility and outcome: a learning classifier system approach. *Journal of the American Medical Informatics Association*, 2013.

[15] R. J. Urbanowicz, A. Granizo-Mackenzie, and J. H. Moore. An analysis pipeline with statistical and visualization-guided knowledge discovery for michigan-style learning classifier systems. *Computational Intelligence Magazine, IEEE*, 7(4):35–45, 2012.

[16] R. J. Urbanowicz, J. Kiralis, J. M. Fisher, and J. H. Moore. Predicting the difficulty of pure, strict, epistatic models: metrics for simulated model selection. *BioData mining*, 5(1):1–13, 2012.

[17] R. J. Urbanowicz, J. Kiralis, N. A. Sinnott-Armstrong, T. Heberling, J. M. Fisher, and J. H. Moore. Gametes: a fast, direct algorithm for generating pure, strict, epistatic models with random architectures. *BioData mining*, 5(1):16, 2012.

[18] R. J. Urbanowicz and J. H. Moore. Learning classifier systems: a complete introduction, review, and roadmap. *Journal of Artificial Evolution and Applications*, 2009:1, 2009.

[19] R. J. Urbanowicz and J. H. Moore. The application of michigan-style learning classifiersystems to address genetic heterogeneity and epistasisin association studies. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 195–202. ACM, 2010.

[20] S. W. Wilson. Classifier fitness based on accuracy. *Evolutionary computation*, 3(2):149–175, 1995.