Published in final edited form as:
Evol Intell. 2013 November ; 6(2): . doi:10.1007/s12065-013-0092-0.

A Multi-Core Parallelization Strategy for Statistical Significance Testing in Learning Classifier Systems

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

Jason H. Moore, and
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-based statistics for evaluating the significance of class prediction, predictive attributes, and patterns of association have only appeared within the learning classifier system (LCS) literature since 2012. While still not widely utilized by the LCS research community, formal evaluations of test statistic confidence are imperative to large and complex real world applications such as genetic epidemiology where it is standard practice to quantify the likelihood that a seemingly meaningful statistic could have been obtained purely by chance. LCS algorithms are relatively computationally expensive on their own. The compounding requirements for generating permutation-based statistics may be a limiting factor for some researchers interested in applying LCS algorithms to real world problems. Technology has made LCS parallelization strategies more accessible and thus more popular in recent years. In the present study we examine the benefits of externally parallelizing a series of independent LCS runs such that permutation testing with cross validation becomes more feasible to complete 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 as long as the number of concurrent processes does not exceed the number of CPU cores, the speedup achieved is approximately linear.

General Terms

Algorithms; Performance; Design

Keywords

LCS; significance testing; parallelization; scalability; multi-core processors

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 [11]. Previously, we introduced a promising new methodology to address these complexities using a Learning Classifier System (LCS) algorithm [13]. Learning classifier

systems (LCSs) [17] 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 [13] we applied our own extended supervised-learning classifier system, called AF-UCS [12], and a statistical and visualization guided knowledge discovery pipeline [14] to a real world genetic epidemiology study of bladder cancer susceptibility. As a result, 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 [4].

To date, formal significance testing of LCS run statistics, such as test accuracy, has been limited to the aforementioned study [13]. One of the major goals of the LCS research community is to extend these unique and powerful algorithms to real world applications. In the context of real world classification and data mining tasks one of the first questions that should be asked when reviewing results is: what is the confidence (i.e. probability) that these results are real or meaningful? Formalized evaluation of confidence avoids subjective biases such as: what constitutes a large sample size?, or what is considered to be small enough variance that we believe our findings to be true? To address these questions, we adopt permutation testing which boasts a number advantages as a strategy for formal significance testing. First, a permutation test can be applied to any statistic, whether or not it's distribution is known. Second, they overcome the problems introduced by unbalanced datasets (i.e. permutation testing can differentiate whether high testing accuracy is meaningful, or the product of a large class imbalance.) Lastly, the generation of a dataset-specific permutation as opposed to a theoretical distribution (such as found in parametric statistics such as the student's t-test) accounts for other potential spurious relationships specific to the the dataset being analyzed. Notably, even formal significance testing is imperfect, since it relies on probability and the traditional yet arbitrary 95% significance cutoff. Regardless, communicating LCS results in the context of the formalized p-value standard is a critical step towards making LCS algorithms accessible, interpretable, and comparable within real world applications.

In order to obtain statistical significance measures in [13], *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 [13], 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. Figure 1 illustrates the combination of cross validation and the permutation test as applied in [13]. Without access to large scale multi-processor clusters (i.e. completed serially on a single 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 [9], (2) fitness calculation in BioHEL, and (3) prediction computation (also in XCS) [10]. 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 [8, 7, 6], we use the multiprocessing [2] 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 and Linux 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 [12]. AF-UCS (attribute feedback UCS), is an expanded and modified implementation of UCS [5]. UCS, or the sUpervised Classifier System, is a michigan style LCS based largely on the popular XCS algorithm [19] 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 [18].

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 sub-sumption on. Parallelization code was incorporated into the Main.py class. The implementation described above is available on request (ryanurbanowiczgmail.com) and will be posted on the LCS and GBML Central webpage [1].

2.2 Implementation

We begin our discussion of implementation with some background. Generally, a process is a collection of data and instructions. These instructions allow the CPU to perform operations on the associated memory in order to complete a task. Processes can range in complexity from simply writing keyboard presses on screen to retrieving data from an HTML server and displaying the formatted data in a web browser. In modern computing, hundreds of processes may be organized by the operating system (OS) at one time. Users often desire or require concurrent execution of CPU tasks such as: loading web pages, playing music or videos, perform word processing, etc. On a machine with one CPU, the OS provides users with the illusion of concurrently executing these processes through clever time slicing. A portion of the instructions in one process will be executed then placed back in a queue while a portion of another process is executed. It is this intelligent juggling of processes by the OS that enables most concurrent execution. The process juggling is more formally called context switching and can be a costly, in terms of time, endeavor. If the processes between which the OS is switching both require large amounts of data in order to perform their instructions, the context switch can be slow. Threads, which are subunits of processes, alleviate this to some degree by enabling light weight context switching. A thread, similar to a process, is a collection of data and instructions. However, threads belong to a process and have access to their parent process' memory. In cases where concurrent tasks should be carried out on relatively constant data, context switching between threads requires little memory management and simply instructs the CPU to juggle lists of instructions. While this is clearly true for machines with a single CPU, it also extends to machines with multiple CPUs. So long as the number of processes or threads being organized by the OS exceeds the number of CPUs available, context switching will take place.

The multithreading framework is a good fit for simple parallel versions of AF-UCS. Specifically, in the case of k-fold cross validation testing, k threads can be initialized to perform model training and testing and all k threads can access the same data belonging to their parent process. The same can be argued for permutation testing; p threads (one per permutation) can be initialized all accessing the same data with the only distinction being changes in the sample labels. Both of these AF-UCS parallelizations would benefit from the light weight context switching that would be provided by a multithreading approach to parallelization. Unfortunately, multithreading in Python is hampered by the global interpreter lock (GIL). The GIL is a form of memory access lock that prevents multiple python threads from running at the same time. In the case of a single CPU, the GIL does not present a problem for a multithreaded implementation of AF-UCS. Each thread would be juggled by the OS and the context switching between them would be relatively light weight. However, in the presence of multiple CPUs the GIL ensures that only one thread can be executed at a time; this means that if an AF-UCS thread is executing on CPU1, another AF-UCS thread cannot simply be executed on CPU2 or any other available CPUs. Effectively, the GIL removes the benefit of the increased number of available compute hardware.

To circumvent the GIL and more effectively capitalize on the common multi-core hardware in modern workstations, we implement our parallel AF-UCS using the Python multiprocessing package. Using this package we forego the creation of threads for CV and

permutation testing and instead create processes. Each process launched contains not only unique CPU instructions but a copy of the data necessary to carry out those instructions. Relative to multithreading, the time necessary to launch a process is greater than that necessary to launch a thread and the context switching is similarly slower. Despite this, we expect a greater overall performance increase as processes are not regulated by the GIL and are free to make use of whatever compute units are available.

Our parallelization of AF-UCS hinges on the 'Pool' object provided by the 'multiprocessing' package in Python (v 2.6 and greater) [2]. 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 [3] or the 'time' command in the Linux Bash shell. Evaluations were completed on four separate workstations and all tests were completed after a fresh boot of the respective workstation. The hardware specifications for each machine are listed in Table 1. On each machine we ran three distinct LCS analysis scenarios: (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.

Algorithm 1

Pseudo Code for CPU Parallelization

```
Require: data, nGen, k, nPermutations, nProcesses
  jobPool = Pool(nProcesses)
  partitions = Partition(data, k)
  for part ∈ partitions do
    jobPool.add(LCS(part, nGen))
  end for
  for permute ∈ nPermutations do
    data = Permute(data)
    partitions = Partition(data, k)
    for part ∈ 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
```

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 [16]. 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 [15]. 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

Four consumer computers were used to perform testing. The hardware details of these workstations are listed in Table 1. These machines possess a range of multicore CPUs which allow the assessment of scaling and performance on hyper-threaded and non-hyperthreaded hardware.

3. RESULTS AND DISCUSSION

As shown in Figure 2, our parallelized algorithm scales approximately linearly up to the number of cores in representative consumer level computers. All computers used for testing had quad-core processors and the speedup achieved using 1 to 4 processes is approximately linear. The Intel Core i7 processors (Desktop 1, Desktop 3, 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 four analysis scenarios with the exception of the CV scenario in which performance deteriorates with the addition of hyperthreaded processes. The scaling of performance was best on Desktop 1 which achieved at least a 4-fold increase in performance in all 3 analysis scenarios (i.e. CV, permutation testing, and CV combined with permutation testing). Across all four work- stations, the CV combined with permutation testing analysis showed the most linear scaling. This was the longest running analysis and the improved scaling suggests that initialization and synchronization costs were overshadowed by the LCS computation.

Figure 3 illustrates the efficiency of our parallel implementation across the different analysis 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 3 are consistent with those in Figure 2, 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 Desktop 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. A similar pattern of performance is seen in the results for Desktop 3. Relative to a single process, using eight processes resulted in an approximate 4 fold decrease in wall-time compared to a 3.8 decrease when using four processes. Though the difference in wall-time between four and and eight process benchmarks was minimal (2.7 minutes), we expect the difference to be much greater in more practical analysis situations with larger input data, population size, and number of generations. More importantly, our results indicate consistent performance across a range of hardware and operating systems; both Windows 7 and Ubuntu 13.04 analysis workstations showed similar patterns in scaling and efficiency. The results support a simple operational heuristic of number of processes equal to the total number of concurrent processes available (including hyperthreading).

While successful, it should be noted that the major bottle-neck of the current implementation of parallel AF-UCS is the final synchronization between all of the processes. We terminate each process with a write to the hard drive in order to save the process results. Once all the processes have completed, AF-UCS processes the output file on the hard drive and tabulates the average testing accuracies and p-values. This bottleneck could greatly be relieved through the use of a data structure in main memory. Access to this data structure would be controlled by a memory lock but we expect communication with main memory to be faster than reading and writing to the hard drive.

4. CONCLUSIONS

This study extends previous work by Urbanowicz *et. al*. [14, 13] 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 focused our implementation evaluation on Windows 7 and Ubuntu workstations. While this workstation selection was based largely on our own hardware availability, the similar patterns of performance across operating systems suggest that our results are robust and should generalize to most user hardware.

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.

Acknowledgments

This work was supported by NIH grants LM011360, LM009012 and LM010098.

REFERENCES

1. Genetics based machine learning central. http://gbml.org/.
2. Python multiprocessing module. http://docs.python.org/2/library/multiprocessing.html.
3. Microsoft powershell. 2012 http://technet.microsoft.com/en-us/library/bb978526.aspx.
4. Bacardit J, Llorà X. Large-scale data mining using genetics-based machine learning. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery. 2013; 3(1):37–61.
5. Bernadó-Mansilla E, Garrell-Guiu JM. Accuracy-based learning classifier systems: models, analysis and applications to classification tasks. Evolutionary Computation. 2003; 11(3):209–238. [PubMed: 14558911]
6. Binet, S.; Calafiura, P.; Snyder, S.; Wiedenmann, W.; Winklmeier, F. Journal of Physics: Conference Series. Vol. volume 219. IOP Publishing; 2010. Harnessing multicores: Strategies and implementations in atlas; p. 042002
7. Foley SS, Elwasif WR, Bernholdt DE. The integrated plasma simulator: A flexible python framework for coupled multiphysics simulation. PyHPC 2011: Python for High Performance and Scientific Computing. 2011
8. Friborg RM, Bjørndalen JM, Vinter B. Three unique implementations of processes for pycsp. Communicating process architectures. 2009; 2009:277–292.
9. Lanzi, PL.; Loiacono, D. Learning Classifier Systems. Springer: 2010. Speeding up matching in learning classifier systems using cuda; p. 1-20.
10. Loiacono, D. Proceedings of the 13th annual conference companion on Genetic and evolutionary computation. ACM; 2011. Fast prediction computation in learning classifier systems using cuda; p. 169-170.
11. Moore JH, Asselbergs FW, Williams SM. Bioinformatics challenges for genome-wide association studies. Bioinformatics. 2010; 26(4):445–455. [PubMed: 20053841]
12. Urbanowicz, R.; Granizo-Mackenzie, A.; Moore, J. Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference. ACM; 2012. Instance-linked attribute tracking and feedback for michigan-style supervised learning classifier systems; p. 927-934.
13. Urbanowicz RJ, Andrew AS, Karagas MR, Moore JH. 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

14. Urbanowicz RJ, Granizo-Mackenzie A, Moore JH. An analysis pipeline with statistical and visualization-guided knowledge discovery for michigan-style learning classifier systems. Computational Intelligence Magazine, IEEE. 2012; 7(4):35–45.

15. Urbanowicz RJ, Kiralis J, Fisher JM, Moore JH. Predicting the difficulty of pure, strict, epistatic models: metrics for simulated model selection. BioData mining. 2012; 5(1):1–13. [PubMed: 22297131]

16. Urbanowicz RJ, Kiralis J, Sinnott-Armstrong NA, Heberling T, Fisher JM, Moore JH. Gametes: a fast, direct algorithm for generating pure, strict, epistatic models with random architectures. BioData mining. 2012; 5(1):16. [PubMed: 23025260]

17. Urbanowicz RJ, Moore JH. Learning classifier systems: a complete introduction, review, and roadmap. Journal of Artificial Evolution and Applications. 2009; 2009:1.

18. Urbanowicz, RJ.; Moore, JH. Proceedings of the 12th annual conference on Genetic and evolutionary computation. ACM; 2010. The application of michigan-style learning classifiersystems to address genetic heterogeneity and epistasisin association studies; p. 195-202.

19. Wilson SW. Classifier fitness based on accuracy. Evolutionary computation. 1995; 3(2):149–175.

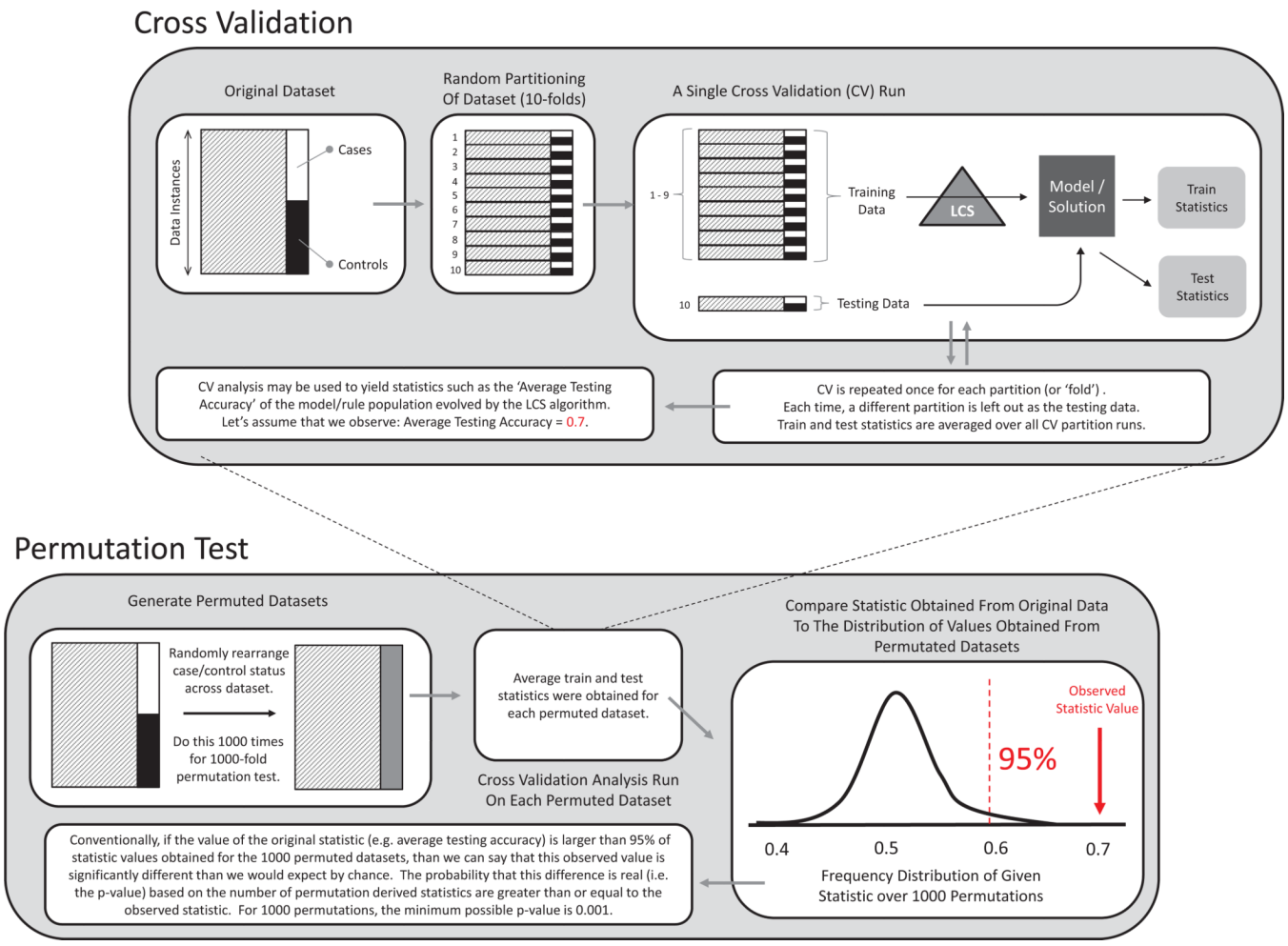

Figure 1.

An illustration of cross validation and the permutation test used simultaneously to obtain train and test statistics along with associated p-values.

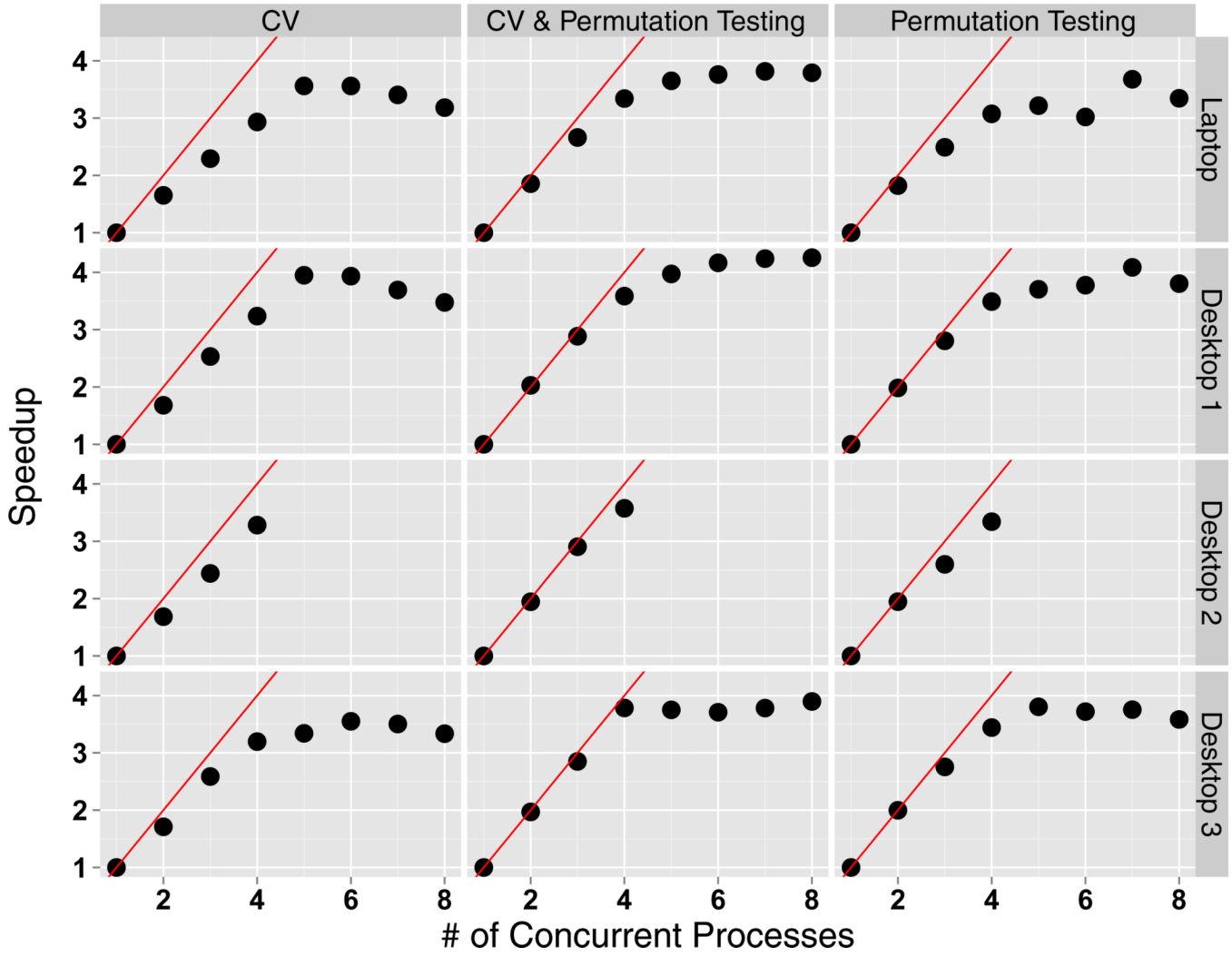

Figure 2.
Speedup of parallelized AF-UCS by number of processes. Red lines indicate ideal linear scaling. Performance is approximately linear up to threads equal to the number of physical CPU cores.

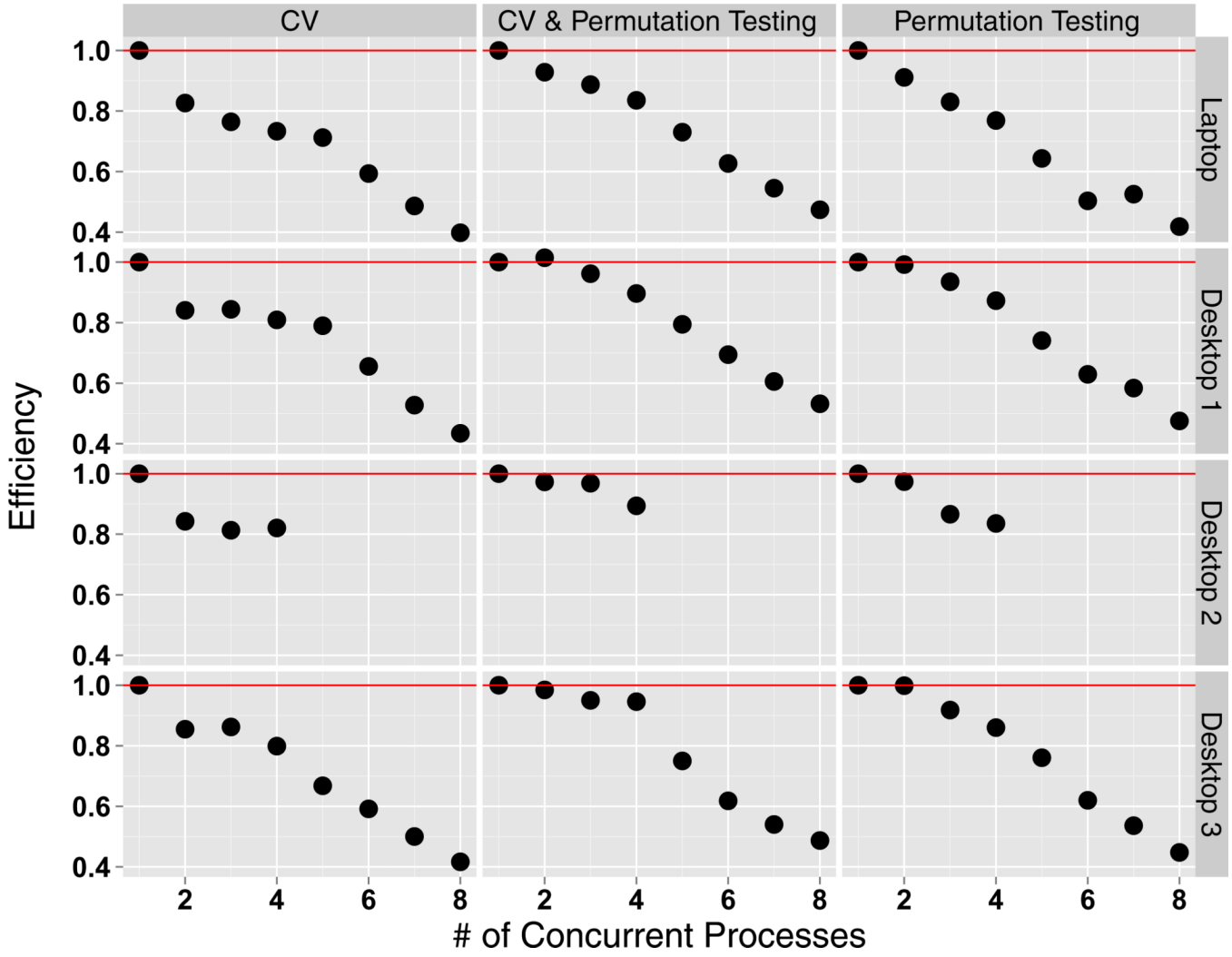

Figure 3.
Efficiency of parallelized AF-UCS by number of processes. Red lines indicate ideal efficiency. Efficiency exceeds 80% up to threds equalling the number of physical CUP cores.

Table 1

Hardware Specifications

Part	Laptop	Desktop 1	Desktop 2	Desktop 3
CPU	Intel Core i7-3840QM	Intel Core i7 950	Intel Xeon E3-1225	Intel Core i7-3770
RAM	16 GB DDR3 798 Mhz	12GB DDR3 1600Mhz	8GB DDR3 667 MHz	16GB DDR3 1600 MHz
Hard Drive	Samsung PM830 477 GB SSD SATA Gen 3.0 6Gb/s	Western Digital 7200 RPM 1TB SATA 3Gb/s	Samsung PM830 238GB SSD SATA Gen 3.0 6Gb/s	Samsung 840 120GB SSD SATA Gen 3.0 6Gb/s
Cores	4	4	4	4
Hyperthreading	Yes	Yes	No	Yes
Processes	8	8	4	8
OS	Windows 7 ×64	Windows 7 ×64	Windows 7 ×64	Ubuntu Linux ×64
Python	2.7	2.7	2.7	2.7