MITHRA: Multiple data Independent Tasks on a Heterogeneous Resource Architecture

Reza Farivar, Abhishek Verma, Ellick M. Chan, Roy H. Campbell
Department of Computer Science
University of Illinois at Urbana-Champaign
201 N Goodwin Ave, Urbana, IL 61801-2302
{farivar2, verma7, emchan, rhc}@illinois.edu

***Abstract*—With the advent of high-performance COTS clusters, there is a need for a simple, scalable and fault-tolerant parallel programming and execution paradigm. In this paper, we show that the popular MapReduce programming model can be utilized to solve many interesting scientific simulation problems with much higher performance than regular cluster computers by leveraging GPGPU accelerators in cluster nodes. We use the Massive Unordered Distributed (MUD) formalism and establish a one-to-one correspondence between it and general Monte Carlo simulation methods. Our architecture, *MITHRA*, leverages NVIDIA CUDA technology along with Apache Hadoop to produce scalable performance gains using the MapReduce programming model. The evaluation of our proposed architecture using the Black Scholes option pricing model shows that a MITHRA cluster of 4 GPUs can outperform a regular cluster of 62 nodes, achieving a speedup of about 254 times in our testbed, while providing scalable near linear performance with additional nodes.**

I. Introduction

The availability of fast commodity GPUs has spawned new interest in using their capabilities to accelerate the state of the art in scientific computing – large-scale computational problems can be solved on desktops rather than using small clusters. Although this innovation has made strides towards bringing cluster-level computational power to the desktop, the approach thus far has not been shown to scale. In this paper, we introduce a methodology to leverage the power offered by these GPUs and the scalability of *MapReduce*[1] clusters. MapReduce is a programming idiom developed by Google for large-scale parallel computations. *Map* and *Reduce* are derived from functional programming where the "Map" function maps an input value to a list of intermediate key/value pairs $Map(k_1, v_2) \rightarrow List(k_2, v_2)$, and *Reduce* produces a collection of values for pairs with the same key $Reduce(k_2, list(v2)) \rightarrow list(v3)$.

We perform a study on *Hadoop*[2] to quantify the benefits of our approach and achieve a **254x** speed up in our testbed. Our studies using several test benchmarks show that there exists an overhead when executing scientific computing algorithms on Hadoop clusters. We traced the observed overhead back to the use of hard disks to store the intermediate key/value pairs created by the mapper tasks and the involved network overhead as dictated by the Hadoop design.

Scientific computing problems are typically bound by available computing cycles rather than I/O activity, in contrast to data processing tasks typically found in search engine companies. MapReduce and Hadoop perform well in I/O-bound problems, for example *word count, inverse indexing* and *distributed grep*, where several simple operations are repeatedly performed over large data sets. The data in these applications is so large that it must be stored on hard disks. Moreover, there is not much reuse of the data, as few operations are performed on each data block. Thus, there does not exist much locality that can be exploited and there would not be any advantage of working with data explicitly in memory. Even though this might be an over simplification, it is not far from typical use cases. Given that scientific computing does exhibit locality in many cases, it would be advantageous to provide provisions in MapReduce frameworks to support in memory storage of intermediate key/value pairs.

In 2004, Philipp Colella described seven scientific computing core kernels, the so-called seven dwarfs of parallel programming[3], that can represent a majority of the scientific computing algorithms. These "dwarfs" include the following algorithm classes: *Dense Linear Algebra, Sparse Linear Algebra, Spectral Methods (FFT), N-body methods, Structured Grids, Unstructured Grids and Monte Carlo.* The Monte Carlo method is often used when the model is complex, nonlinear, or involves more than just a couple uncertain parameters[4]. Monte Carlo methods are versatile enough to cover a wide problem domain including the simulation of galactic formation to business risk analysis to solving systems of linear equations[5]. In this paper, we show that the last problem

class can map very well onto a cluster of computers with GPUs, programmed using the MapReduce programming model. From a theoretical point of view, the Monte Carlo dwarf is equivalent to the MapReduce model. However, in practice, one often encounters secondary effects which might adversely affect performance on different architectures. MapReduce, as discussed in this paper, can readily be deployed on a commodity cluster of workstation-class machines sporting moderately priced GPU cards. From our experiments, we demonstrate that this architecture can perform very well on processing-intensive Monte Carlo jobs. An important factor in how good a Monte Carlo class problem can be mapped to a cluster of GPU machines is the degree of associativity and commutativity of the reduction function. In Section II, we elaborate further on these properties.

The rest of this paper is organized as follows: Section II explores the mathematical properties of MapReduce and MUD algorithms and show a direct correspondence with the Monte Carlo method in Section III. Section IV introduces the *MITHRA* architecture and Section V evaluates the performance of our work with respect to the Black Scholes Option Pricing model and describes how we have mapped the various portions of this model to a Monte Carlo simulation over multiple implementations on stand alone machines and clusters. We then compare the performance results against MITHRA, our implementation of MapReduce running on CUDA technology linked together with MapReduce. Section VI explores related work and we conclude with Section VII.

II. MapReduce, MUD and Monte Carlo

In this section, we study the mathematical properties of MapReduce in the context of Monte Carlo simulations. We first introduce MapReduce, then we define the associated operations mathematically and show that the corresponding parts are also present in Monte Carlo simulations, thus establishing a direct relationship between the two methods.

A. Theoretical foundations of MapReduce model

The MapReduce framework has shown considerable prowess in the area of data-intensive computing and shows promise in processor-intensive scientific computing as evidenced by the application of the Monte Carlo dwarf. Our work formally shows the relationship between MapReduce and Monte Carlo.

In this paper we borrow the formalism provided in [6], which introduced a simple formal model for the class of algorithms that can be programmed with the MapReduce model, referred to as *MUD* (Massive Unordered Distributed) algorithms, with slight modifications.

A MUD algorithm is formally defined as a 6-tuple mathematical structure $m = (\Phi, \oplus, \eta, \Sigma, (K,V), \Gamma)$. $\Phi, \oplus$ and η are functions that the framework programmer provides; $\Sigma, (K,V)$ and Γ are sets acting as the domain and image sets of the former functions. Members of the (K,V) set are tuples of the form (k_i, v_i).

The function $\Phi : \Sigma \rightarrow (K,V)$ can map an input item σ from its domain set Σ to a list of $(key, value)$ tuples, where the list can contain 0 or more items. The Φ function corresponds to the "map" function written in Google's MapReduce framework or Hadoop, and (K,V), being the image set of Φ, is the intermediate key/value domain.

The aggregator $\oplus : (K,V) \times (K,V) \rightarrow (K,V)$ is a binary operator that maps two items from the set (K,V) to a single item as follows: $(k, v_1) \oplus (k, v_2) \rightarrow (k, v_3)$. The $\oplus$ aggregator therefore "reduces" the intermediate key/value pairs to a single key/value pair for each key. Notice that we do not limit the $\oplus$ aggregator to be commutative and associative, unlike the formalism in list homomorphisms for the monoid used for reduction[7]. However, the output can depend on the order in which $\oplus$ is applied.

Let T be an arbitrary binary tree with n leaves. We use $m_T(x)$ to denote the $(k, v) \in (K,V)$ that results from applying $\oplus$ to the sequence $\Phi(x_1), ..., \Phi(x_n)$ along the topology of T with an arbitrary permutation of these inputs as its leaves. Notice that T is not part of the algorithm definition, but rather, the algorithm designer needs to make sure that $m_T(x)$ is independent of T. This is implied if $\oplus$ is associative and commutative; however, this is not necessary.

The optional post-processing operator $\eta : (K,V) \rightarrow \Gamma$ produces the final output. The overall output of the MUD algorithm is then $\eta(m_T(x))$, which is a function $\Sigma^n \rightarrow \Gamma$, where Σ^n means the Φ function is applied on n members of the Σ set. We say that a MUD algorithm computes a function f if $\eta(m(\cdot)) = f$ for all trees T, in other words, the intermediate key/value pair ordering for each key would not change the outcome of the computation.

B. Parallelism in MUD formalism

The main advantage of the MUD formalism is the broad range of problems that can be described and modeled using it, and the potential for parallelism. There are four distinct parallelism opportunities in a MUD algorithm, which are as follows:

1) Input data set creation
2) Data independent execution of the Φ function
3) Intra-key parallelism of $\oplus$
4) Inter-key parallelism of $\oplus$

We defined the input values of a MUD algorithm to be members of the set Σ. However, in real applications, its members should either be created at run-time, streamed into the machine executing the algorithm or stored on the hard disks distributed across the nodes of a cluster. Each of these possible cases allow for a parallelism and hence better performance. Considering the typical I/O latency that dominates the latter two classes, the best parallelism opportunities can be found when the data can be generated right before consumption. It should be noted that the input data set creation can also be another MUD algorithm, starting with a simpler input data set (for example constant values).

The second class of parallelism is derived from the definition of the Φ function. Note that the input to each evaluation of Φ is an independent, single member of Σ (Note that each $\sigma \in \Sigma$ can be a vector of values). Therefore, there is no dependency among multiple executions of Φ, and it can be parallelized among whatever number of processing elements available on the physical machine executing the algorithm.

The latter two types of parallelism are application dependent, contingent on the problem being solved and the mathematical properties of $\oplus$. Inter-key parallelism can be used when the programmer uses multiple keys in the intermediate key/value pairs. Recalling from the definition of $\oplus$ function, $(k, v_1) \oplus (k, v_2) \rightarrow (k, v_3)$. Therefore $\oplus$ works on multiple values of key/value pairs with the same key, and generates another key/value pair with that key. If the programmer defines the program in such a way to have multiple keys, each execution of $\oplus$ is limited to the key/values with a single key. This parallelism can be used to gain performance in a cluster setting even if there is no intra-key parallelism to be found. A simple yet effective approach would be to have the same amount of keys as the nodes in the cluster, and distributing the total members of Σ among them with different keys.

The last type of parallelism potential, intra-key parallelism, can be exploited if the $\oplus$ aggregator is associative and commutative. In this case, the evaluation of $\oplus$ on a set of key/value pairs (with the same key) can be performed as a binary tree reduction. This operation can therefore theoretically be finished in $O(log(n))$ time, if the number of processing elements is at least $\frac{n}{2}$. If the number of processing elements is less than that ($k < \frac{n}{2}$), the dominant execution time will be the time to evaluate $\oplus$ function for pairs of leaves $\frac{n}{2}$ times, which can be parallelized across k processing elements. Therefore, the execution time can be reduced to $O(\frac{n}{2 \cdot k})$.

Sometimes, even if the $\oplus$ aggregator is not commutative and associative, one can decompose it into the composition of two functions: one that is associative and commutative, and one that is not. Then, it becomes possible to integrate the non-commutative or non-associative part with the η post processing function, and therefore change $\oplus$ into a well-behaved binary operator. As an example, consider the mean aggregator $*$ as $a * b = \frac{a+b}{2}$. We know that this function is not associative, since $(a * b) * c = \frac{\frac{a+b}{2}+c}{2} \neq \frac{a+\frac{b+c}{2}}{2} = a * (b * c)$. However, considering the distributivity property of the division operator, we can decompose the averaging function and leave the division for the last step, performing it in the $\eta(\cdot)$ function.

III. Monte Carlo simulation

Monte Carlo methods can be loosely considered statistical numerical simulation methods where sequences of random numbers are used to perform the simulation. These statistical simulations methods differ from conventional numerical discretization methods which are used to analyze ordinary or partial differential equations that model physical or mathematical systems[8].

Monte Carlo simulation requires the system to be either described by probability distribution functions or a parametric model. Informally, this means that any implementation of a Monte Carlo needs to create volumes of high quality random numbers and use these numbers as inputs to a parametric model of the system. Simulations are performed over many trials and the desired result is filtered through an aggregation function which combines the output of the trials. Typically, one would like to use the average and variance functions to perform this aggregation and the number of samples is chosen to achieve a given accuracy level of the simulation.

The following steps define the required components of a typical Monte Carlo simulation[4]:

1) Create a parametric model, $y = f(x_1, x_2, ..., x_q)$
2) Generate a set of random inputs, $x_{i1}, x_{i2}, ..., x_{iq}$
3) Evaluate the model - and store the results as y_i
4) Repeat - steps 2 and 3 for $i = 1$ to n
5) Analyze the results - using histograms, summary statistics, etc.
6) Error estimation - an estimate of the statistical error (variance) as a function of the number of trials and other quantities can be determined as part of this step.

After the random trials are complete, the results must be aggregated over a combiner function to produce the desired result for the particular problem. However, the essence of the Monte Carlo method is the use of random

sampling techniques to approximate the solution of a complex problem[8].

A. Monte Carlo simulation as a MUD algorithm

The first step of the Monte Carlo algorithm described in the previous subsection is typically done in the design stage, and does not correspond to a certain algorithmic step. The important thing to note however, is that the function f has a limited set of arguments, x_1 to x_n, and the input data set of each trial is independent of other trials. Described as a MUD algorithm, this application and domain specific function will form the function Φ. The next step of the typical Monte Carlo algorithm requires an efficient method of creating $n \cdot q$ random, psuedo-random or quasi-random numbers, in other words creating the Σ set. Formally, each $\sigma \in \Sigma$ is a vector of q elements: $\sigma = (x_1, x_2, ..., x_n)$, and Σ is the set of all random vectors of size q.

The third and fourth steps of the Monte Carlo application are about evaluating the model. In other words, this is where the Φ function is applied in parallel to all the random input vectors. Recalling that in a Monte Carlo simulation all of the trials will contribute equally to the final summary statistic values of interest, and hence the requirement of considering all of the trials in calculating the final analysis results, we can simply use the same key for all the key/values pairs generated here in a MUD model. Formally, $\Phi(\sigma_i) = (k, v_i)$. It should be noted that the MUD formalism allows to create a list of key/value pairs by Φ, but the typical Monte Carlo algorithm uses each input random vector once to create a single value. Also note that the input values to each round of function evaluation, conforming to MUD formalism requirements, are independent and thus allowing the function evaluations to be performed in parallel.

The final step of the Monte Carlo simulation algorithms involves applying summary statistics to the n evaluated trials of the function Φ. Since in the previous step, all of the intermediate key/value pairs were generated with a single key, there is no intra key parallelism. However, many of the summary statistics functions are either associative and commutative, or can be decomposed into two functions, Φ and η so that Φ captures the main functionality while being associative and commutative, and η function which is not commutative or associative, performing the final operation. Examples of such functions include mean and variance calculation. Therefore, the summary statistics functions can utilize the inter-key parallelism of a MUD algorithm as mentioned earlier.

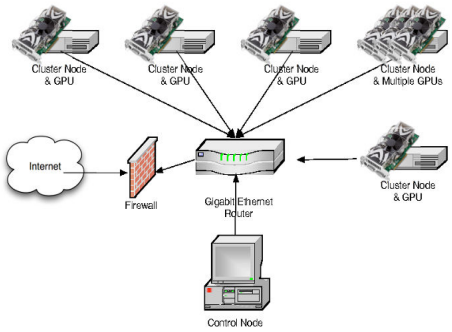

Fig. 1. Architecture

IV. MITHRA ARCHITECTURE

MITHRA is designed to excel at executing massive, data independent computing tasks. We leverage heterogeneous computing resources in the architecture design. In other words, the "correct" computing resources for each phase of a MUD algorithm can be different, therefore the potential of leveraging different hardware classes. The MITHRA architecture takes the mathematical formalism described in previous sections and reifies the results into an efficient organization of hardware to exploit the properties exposed by our formalism. Specifically, we leverage the four optimizations mentioned in SubSection II-B. In this section, we describe the architecture in detail and explain how it achieves the parallelism goals set out earlier.

A. Design

Clusters of GPU have been utilized in the recent years to solve different classes of problems as discussed in Section VI. Similarly, MapReduce or MUD algorithms have been used for both massive log analysis and to a lesser extent for scientific computing. The novelty of MITHRA, however, is the adoption of the MUD formalism to model a broad range of scientific computing problems, and adapting it to run on a low cost commodity cluster of computers. In its basic design, MITHRA is a cluster of COTS computing nodes. Each node contains a mid-range CPU and is connected to other nodes through a gigabit-class ethernet network. Graphics Processing Cards (GPUs) installed on each processing node make MITHRA suitable for running MUD algorithms.

Figure 1 depicts the architecture of MITHRA. The MITHRA framework is based on the open source Hadoop project, which is an implementation of Google MapReduce. It inherits the merits of fault tolerance and scalability from Hadoop and speed from GPUs. It

should be noted that in a cluster of COTS components, the probability of component failures is considerably higher than a cluster of specialized high performance and highly reliable components. However, the adoption of MapReduce programming model implies software fault tolerance features that guarantee the continued execution of computation jobs even under less than ideal conditions, without the programmer having to deal with them.

The Map function in MITHRA is designed as a two stage process. The first phase makes use of the Hadoop Map function. The Hadoop framework distributes the user supplied map function across the cluster. However, the main functionality of the the Φ function in the MUD formalism is not programmed in the Hadoop Map. Hadoop's Map merely acts as a distribution mechanism for the Φ workload across the nodes of the clusters.

The Φ function in a MUD algorithm, being data independent in its multiple evaluations on its input data set Σ, can be performed on GPUs. This is the second phase of the Map function design in MITHRA, in which the programmer writes a CUDA kernel to work on one $\sigma \in \Sigma$. In fact, the data independence property of the Φ function in the MUD formalism is similar and matches the data-independent compute intensive SIMD[9] kernel model of CUDA[10]. Unlike a traditional Map function in the MapReduce model, MITHRA does not promote having k Maps with each Map evaluating the Φ function $\frac{N}{k}$ times, where N is the number of data items in Σ and k the number of nodes in the cluster. Instead, the map function written in CUDA corresponds directly to the function Φ, working on a single item from Σ.

The Hadoop framework distributes Σ among the nodes of the cluster. The Hadoop map task divides up the amount of data determined to be processed on that node to chunks of data small enough to fit in the GPU memory (typically 64MB to 256MB in our implementation). It then loads the data in the GPU memory and calls the CUDA kernel.

The next part of the framework involves intermediate key/value management. Focusing on the Monte Carlo simulation class of problems, we did not need to use more than a single key for the intermediate key/value pairs, therefore the implementation of the key/value management scheme is not completely general. As this is an important feature for the application to more general problems, it is one of our first future work milestones. In our current implementation, after the execution of the Map functions, the intermediate key/value pairs are assumed to have been grouped by *the key*.

The last phase of a MUD algorithm is the application of the $\oplus$ aggregator on all the key/value pairs with the same key, which in a Monte Carlo simulation means all of the intermediate values across the cluster nodes. In a Monte Carlo simulation, the Φ function calculates the mean and variance of the trial runs of ϕ as indicator summary statistics values. As discussed earlier, even though these functions are not associative and commutative, they can be decomposed into separate functions, mostly because the division operator used in mean and variance is distributive. Therefore, the reduction aggregator $\oplus$ can be applied locally and the pre-final values can be sent to the head node for the final round of $\oplus$ and $\eta(\cdot)$ application.

B. Practical implications of adopting MUD programming model on MITHRA

Recalling the formalism presented in Section II and the discussion of the inherent parallelism in MUD algorithms, this subsection shows how each of the four parallelism opportunities can be exploited for better performance in MITHRA. We formally show that all Monte Carlo simulations can be performed in an efficient manner in our MITHRA architecture. Steps of the Monte Carlo algorithm correspond to a phase of the MUD model, or match an inherent parallelism opportunity of MUD.

1) Input Data Set: The input data set can either be pre-stored and distributed on hard disks of the individual machines in the cluster, can be streamed in from the Internet or can be generated as used. When the data is already distributed among the hard disks, all of the Hadoop Map functions can read the input data in parallel. Assuming k nodes are available in the MITHRA cluster, the theoretical aggregate I/O bandwidth becomes k times the I/O bandwidth of each hard disk.

When streaming from the Internet, each node receives its own specific data set. In theory, this task might not be highly parallelizable since the incoming network connection bandwidth becomes the bottleneck. However for practical purposes, it can have enough bandwidth to fill all the node's input data bandwidth specification, and thus the task can be parallelized.

Finally, our best parallelism can be achieved when the application needs or can be changed to use run-time generated values as its input set. This is because even though the generation of the Σ set takes time, it will probably take less time than to load them from hard disks or read them through slow network connections. This is the case for the Monte Carlo simulation algorithms, since they only require a set of random numbers as their Σ set.

We generate quasirandom numbers using the Niederreiter quasirandom generator[11] in the GPUs. A quasirandom or low discrepancy sequence, such as the Faure, Halton, Hammersley, Niederreiter or Sobol sequences[11], [12] is "less random" than a pseudorandom number sequence, but more useful for such tasks as approximation of integrals in higher dimensions and in global optimization. This is because low discrepancy sequences tend to sample space "more uniformly" than random numbers. Algorithms that use such sequences may have superior convergence[11]. Creating quasi or psuedo random numbers is a very time consuming task, therefore it is best to create the random numbers in the GPUs where they will be consumed in a distributed manner to utilize all the processing elements available, in contrast to creating them directly in the CPU using a sequential algorithm and then transferring them to the GPU memory through the low bandwidth PCI-Express bus. We base this part of our work on an implementation of this algorithm from the CUDA SDK[13].

Another requirement for the Monte Carlo simulation algorithms is the application of a normal distribution PDF to the random numbers. For the inverse cumulative normal function $z = N'(p)$, there are several numerical implementations providing different degrees of accuracy and efficiency[14]. A very fast and accurate approximation is the one given by Boris Moro in [15]. This algorithm also runs on the GPU.

2) Data Independent Φ*:* As mentioned earlier, Φ is data independent across executions, and can match the SIMD execution model of GPUs. For our experiments we chose the "Black Scholes" option pricing equation, which conforms to the requirements of the MUD formalism for Φ.

3) Inter-key Parallelism of $\oplus$*:* This parallelism opportunity can be exploited when the application uses more than one key in its intermediate key/value pairs. The inter-key parallelism is usually a side effect of the algorithm used to solve a problem correctly. However, the programmer can use it as a measure to force parallelism across nodes of a cluster, especially when the $\oplus$ aggregator is not associative or commutative.

For a Monte Carlo algorithm, the intermediate keys need not be different, and the reduction aggregator can be made into an associative and commutative function, and therefore the intra-key parallelism (described next) can exploit all the inherent parallelism available in the problem.

4) Intra-key Parallelism of $\oplus$*:* The last step of a Monte Carlo algorithm is the calculation of required indicator summary statistics. We have provided examples and details of two summary statistics functions used in our implementation: mean and variance. Both of these functions are decomposed in our implementation, so that an associative and commutative function (addition) is used as the aggregator. Therefore, this aggregator function can be applied inside the GPUs, and then across multiple nodes of the cluster in the Hadoop reduction function (which acts as the $\eta(\cdot)$ function of the MUD formalism).

V. Evaluation

A. Black Scholes Overview

A financial option[16] is the right, but not an obligation to purchase an asset at a future date(*expiration date*) at an exercise price. *Call options* grant the holder the right to purchase an underlying asset while *Put* options allow the holder to sell. Since the price of the underlying asset varies over time due to volatility, pricing an option is an extremely important process. Theory suggests that if options are traded in the free market, their price will converge to a *fair* market value. Black Scholes is one method of calculating this value by simulating many possible paths where the price drifts up and down along the time axis following a random Brownian motion of gains and losses.

In this paper, we concern ourselves with European stock options which can only be exercised at the expiration date, and we do not consider transaction costs, dividends and restrictions on short selling. We assume that money can be borrowed freely at a risk-free rate.

To compute the price of an option, we apply the mathematical formula shown in Algorithm 1 to a Monte Carlo simulation to estimate possible final option prices. The parameters to this equation are: *S*, which represents the asset value function, *r* represents the continuously compounded interest rate, σ the volatility of the asset and *T* the expiry time. Each trial computes the potential gain from buying the asset at the agreed upon option price and selling it for the fair market value. Given the random normal distribution of points sampled, taking the arithmetic mean and standard deviation of the distribution establishes the exercise price and strike price.

B. Performance

The algorithm is divided into three stages: the Φ stage performs the map and evaluates the Black Scholes formula over random Gaussian sample points producing intermediate key/value pairs. Once the mappers finish executing, the framework collects all the intermediate key/value pairs with like keys and feeds them to the reducer $\oplus$. The final computation is produced by η

which calculates the mean and standard deviation [1] of the samples. To evaluate the performance of our architecture across multiple parallel programming implementations, we implement the following pseudocode on all architectures.

$\Phi(n)$:
  for $i = 1..n$
    $G \leftarrow$ generate a Gaussian random number
    $V \leftarrow S \cdot exp\left((r - \frac{\sigma^2}{2}) \cdot T + \sigma\sqrt{T} \cdot G\right)$
    value $\leftarrow exp(-r \cdot T) \cdot max\{V - E, 0\}$
    emitIntermediate$(1, value)$
    emitIntermediate$(2, value^2)$

$\oplus(key, values)$:
  sum[1] $\leftarrow$ 0
  sum[2] $\leftarrow$ 0
  while($values$.hasNext())
    sum[key] $\leftarrow$ sum[key] + $values$.next().get()

$\eta(sum, n)$:
  mean $\leftarrow \frac{sum[1]}{n}$
  variance $\leftarrow \sqrt{\frac{sum[2]}{n} - mean^2}$

Algorithm 1. Black Scholes as a MUD algorithm

C. Threaded Implementation

In this implementation, we use standard POSIX threads to compute the option price on an dual quad-core Intel E5410 2.33 Ghz machine with 4 Gb memory. To do so, we apply a four-step process:

1) Generate a pool of sample points for evaluation
2) Break the pool of sample points into n subpools, one for each thread
3) Each thread evaluates the Black Scholes formula over all the sample points
4) Mathematically, we average the means and sum the variances to compute the overall mean and standard deviation of the equal-sized buckets

One issue encountered for this implementation was the use of the standard C-library *rand()* function because the function is not re-entrant and thus locking is required to maintain state consistency. Instead, we use the re-entrant *rand_r()* function to avoid contention issues.

Figure 2 depicts the performance of the threaded version of Black Scholes. We simulate the single threaded version of the program by simply running the algorithm with a single thread. In each case, the map portion dominates the computation while the reduce accounts for less than one percent of the total execution time.

[1]The calculation of standard deviation from sum of squared values is optimized as follows: $\sum_{i=1}^{N}(x_i - \bar{x})^2 = \left(\sum_{i=1}^{N} x_i^2\right) - N\bar{x}^2$
$\implies \sigma = \sqrt{\frac{1}{N}\left(\sum_{i=1}^{N} x_i^2\right) - \bar{x}^2}$

D. Phoenix Implementation

Phoenix[17] is a MapReduce framework for multi-core/multiprocessor machines. MapReduce takes a set of inputs, splits it into batches, then processes each input using a mapper. The mapper then emits key value pairs into intermediate storage and the reduce function then sorts all key value pairs and processes each pair with identical keys into a reducer. At the end of the process, a result is emitted. In Phoenix, as in Hadoop (discussed in the next section), the splitter first breaks up the batch of sample points over *n* mappers. Each mapper then evaluates the sampling function at each sample point and collects these intermediate outputs into a key value pair as per the MapReduce framework. The reducers then take the intermediate outputs and processes them into the final output.

In our implementation, we use the same dual quad-core machine and vary the number of mappers. Figure 4 shows the performance of the Phoenix engine plateauing as early as four cores and doubling the processing power seems to have diminishing returns. When operating at full speed, Phoenix beats the multithreaded implementation by about 200 seconds.

E. Hadoop

We implemented Black Scholes on Hadoop (0.19) and ran it on our 496 core (62 nodes) Hadoop cluster[2]. In our design, each mapper is responsible for generating random sample points and evaluating the sampling function at each point. Upon completion, each mapper outputs two $(key, value)$ pairs as intermediate output: $(0, sumValues)$ and $(1, sumSquaresValues)$. We use the Combiner optimization mentioned in [1] where in, we perform local reductions on each node and then emit $(0, sumLocalValues)$ and $(1, sumSquaresLocalValue)$. In the end, we use a single reducer to compute the mean and standard deviation over all the evaluated sample points. However, we did not aggressively optimize the implementation.

The cloud configuration dictates 4 mappers and 4 reducers per node. Each node is a dual quad core with 8 GB of RAM and 292GB of disk space. We run the MapReduce program with 32, 64, 128, 248 and 256 mappers. In each experiment, we perform a total of 4 billion iterations of the Black Scholes algorithm. We observed an increase in the run time when the number of mappers increases from 248 to 256 due to the increase in the overhead of the Hadoop framework and when there are more map tasks than the number of available map slots (62 * 4 = 248).

[2]http://cloud.cs.illinois.edu

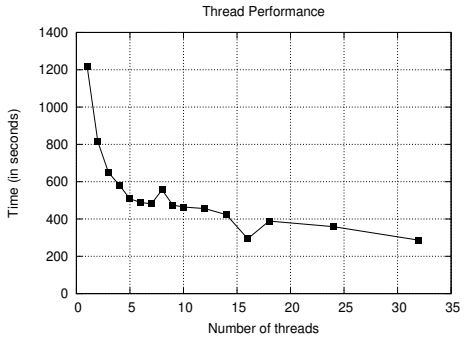

Fig. 2. Thread Performance

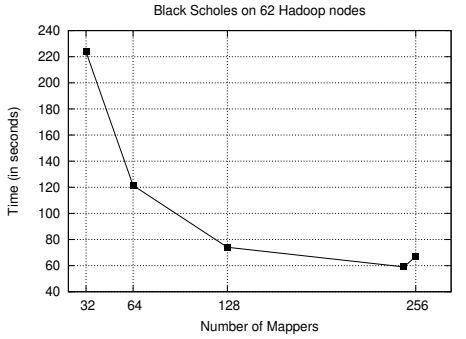

Fig. 3. Hadoop Performance: The number of mappers is the same as the number of cores utilized

F. CUDA Implementation

We also implemented the same Black Scholes calculations on NVIDIA's CUDA[10] general purpose GPU architecture. Rather than using the MARS[18] GPU MapReduce framework, we opted to design our system directly using NVIDIA's framework because it afforded us more control over execution. To ensure a fair comparison, we followed the MapReduce model introduced throughout this paper, where each computing element performs a single Black Scholes calculation in the Map phase. As mentioned earlier in the paper, the required random numbers are created in the GPU using a Niederreiter quasirandom generator[11].

We ran our experiments on two different NVIDIA GPUs. The first card is a 9800 GX2 with 2 GPU chips onboard, each having 128 processing elements running on a core clock of 600 Mhz, and 512 MBytes of 256-bit bus GDDR3 for each chip. The second GPU card is a Quadro FX570 with 16 processing elements running on a core clock of 460 MHz and sporting 256 MBytes of 128 bit DDR2 RAM. The reduction is also programmed to run in the GPU using a binary tree reduction approach.

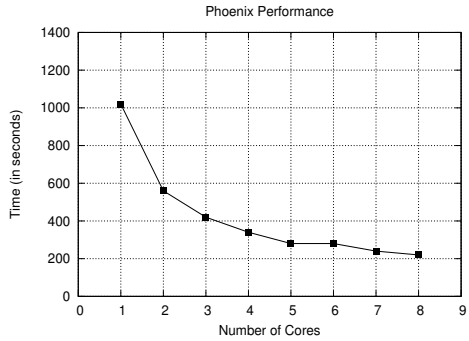

Fig. 4. Phoenix Performance

The first experiment was run on the 9800 GX2. We ran the Black Scholes computation for 4 billion iterations. Since such an array would be larger than the physical memory of our cards, we split the data set into smaller segments, and ran them in sequence. Each segment processes a data set of 64 MBytes, which counting for additional intermediate arrays takes less than 512 MBytes of memory and therefore fits in our GPU card memory. In total, the experiment ran for 24.51 seconds where the map stage finished in 8.34 seconds while the reduce took 16.2 seconds. By leveraging the parallelism inherent in CUDA, the map finishes in a short amount of time while the reduce was inherently less parallel. This is a stark contrast to the multi-core experiments, where the map stage took the majority of run-time. For the Quadro 570 with 16 cores, we reduce the size of the immediate array to 16 MBytes to match the size of the video memory. In this configuration, the total runtime is 245.77 seconds, where the map consumed 108.18 seconds and reduce took 137.59 seconds. The results are shown in Figure 5.

G. MITHRA

MITHRA combines the benefits of Hadoop's distributed computing with CUDA's raw processing power. Our technique requires no change to the Hadoop scheduler or any part of its core. Hadoop streaming[19] allows programmers to specify native binaries as mappers and reducers. Our mapper is a native Linux process which utilizes the unmodified NVIDIA CUDA version of Black Scholes mentioned earlier. Each node then utilizes the full power of its GPUs and performs the mapping process at full steam. Once the mappers complete, the local reducers (like the combiner optimization in Hadoop) begin to process the aggregate results using the binary tree reduction optimization. These intermediate values are then passed through the Hadoop framework to a

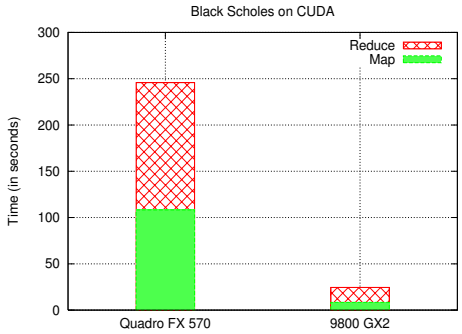

Fig. 5. CUDA Performance on two different NVIDIA GPUs

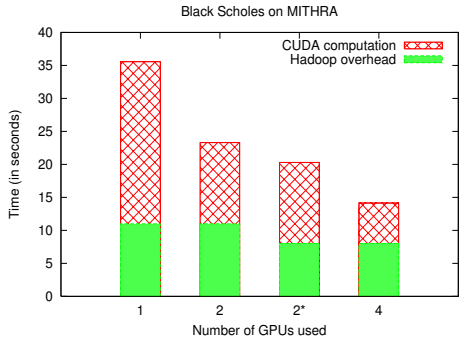

Fig. 6. MITHRA peformance

single reducer which performs the final computation of means and standard deviation over all the evaluated sample points. As shown in Figure 6, we perform four experiments that vary the number of GPUs used running a total of 4 billion iterations of the Black-Scholes algorithm. In the first experiment, only a single node is used in Hadoop and only one CUDA card is utilized. The Hadoop overhead is estimated by using an identity mapper and an identity reducer, which shows the amount of time taken by the Hadoop framework to fetch the inputs, start the program and write the output. Since the 9800 GX2 cards are actually 2 cards in one package (they even have two separate PCBs), and since most of the computation of both Map and Reduce are performed in GPUs, in the next experiment we utilize these 2 GPUs on the same machine. Our assumption here is that since the host CPU is not involved in either Map or Reduce, we can treat each physical host as two nodes of the Mithra cluster. Of course Hadoop has its own overhead, therefore this results in slower completion than if a single GPU is used on 2 machines (shown as 2*). Finally, we run our experiment utilizing both the CUDA cards on both machines.

Comparing the result with those of Hadoop, we see that a MITHRA cluster of 4 nodes runs the total computation in 14.4 seconds, while the Hadoop cluster takes 59 seconds. This means that for each node, the MITHRA architecture has $(59s \cdot 4map \cdot 62nodes)/(14.4s \cdot 4GPUs) = 254$ times performance improvement.

VI. Related Work

MITHRA is based on several well researched ideas and projects. It built upon the Hadoop[2] open source MapReduce framework, and extends it with the ability to run GPGPU kernels[10]. Even though MapReduce[1] and Hadoop are recent developments, their basic mathematical underpinnings were discussed as early as 1987[20], and has been well studied afterwards[21], [22], [23], [24]. List homomorphisms research is typically more theory oriented, however, and to the best of our knowledge has not been utilized in practical applications and architectures thus far. Other formalism also exists for Map Reduce types of computation[25], [6]. Our formalism is an extension of that presented in [6] with some modifications to support the specifics of MapReduce.

The simplicity of the MapReduce model has long been one of it's most attractive features, resulting in several implementations of various flavors and widespread practical use. Aside from typical cluster implementations[1], [2], other implementations of this framework exists for other platforms as well. Mars[18] is another attempt at bringing the MapReduce paradigm to a non-distributed system, implemented on a single NVIDIA G80 GPU. It aims to hide the complexities of GPU programming, while still achieving strong performance. However,[18] does not provide a solution for scalability[26] brought the MapReduce paradigm to heterogeneous multi-core systems. By taking a high level library-based approach, they turn the usually ad-hoc and exhaustive approach to programming for heterogeneous parallel systems into a more manageable one, giving strong scalability with minimal programmer effort.

There has been lots of recent work on clustering GPUs. Fan et al[27] proposed using cluster of GPUs for scientific computing. Their work preceded the availability of general purpose GPU programming platforms, therefore they use GPGPU techniques to masquerade their computation as a graphics application. Zippy[28] abstracts GPU cluster programming with a two level parallelism hierarchy and a non-uniform memory access model. They adapt the Global Arrays programming model to the GPU cluster model and combine it with the stream processing model. A scalable parallel framework

based on MapReduce has been built by Tu et al[29] for analyzing terascale molecular dynamics simulation trajectories. Similary, Phillips et al describe strategies for the decomposition and scheduling of computation among CPU cores and GPUs, and techniques for overlapping communication and CPU computation with GPU for kernel execution for NAMD[30].

VII. Concluding Remarks and Future Work

The MITHRA architecture shows that a set of GPU-enabled nodes running on a MapReduce cluster can achieve significant speedup of scientific computing applications. Using the popular Black-Scholes option pricing model, we achieved a speedup of over 254 times per cluster node. Our mathematical analysis reveals that the speed gains apply not only to MapReduce problems, but also to more generalized MUD domains. These results on the MITHRA architecture demonstrate the merits of our architecture and opens up opportunities to apply parallel computing in a distributed environment. By virtue of increased productivity per node, we are able to significantly reduce the number of nodes required to perform the computations.

VIII. Acknowledgements

We would like to thank the anonymous reviewers for their valuable feedback. This research was supported by grants from Motorola,. Our experiments were performed on hardware generously donated by NVIDIA. This work was funded, in part, by NSF IIS Grant #0841765. The views expressed are those of the authors only.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] "http://hadoop.apache.org/."

[3] P. Colella, "Defining Software Requirements for Scientific Computing," *presentation*, 2004.

[4] J. Wittwer, "Monte Carlo Simulation Basics," Jun 2004. [Online]. Available: http://vertex42.com/ExcelArticles/mc/MonteCarloSimulation.html

[5] D. Landau and K. Binder, *A Guide to Monte Carlo Simulations in Statistical Physics*. New York, NY, USA: Cambridge University Press, 2005.

[6] J. Feldman, S. Muthukrishnan, A. Sidiropoulos, C. Stein, and Z. Svitkina, "On the complexity of processing massive, unordered, distributed data," May 2007. [Online]. Available: http://arxiv.org/abs/cs/0611108

[7] Z. Hu, H. Iwasaki, and M. Takechi, "Formal derivation of efficient parallel programs by construction of list homomorphisms," *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 3, pp. 444–461, 1997.

[8] U. Cherubini and G. D. Lunga, *Structured Finance: The Object Oriented Approach (The Wiley Finance Series)*. John Wiley & Sons, 2007.

[9] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. 21, no. 9, pp. 948–960, September 1972.

[10] "Cuda programming guide," *Available at http://www.nvidia.com/cuda*.

[11] H. Niederreiter, *Random number generation and quasi-Monte Carlo methods*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1992.

[12] *Random number generation and monte carlo method*. Springer-Verlag, 1998.

[13] "Niederreiter quasirandom sequence generator," *Available in NVIDIA CUDA SDK*.

[14] P. Jaeckel, *Monte Carlo Methods in Finance*. Wiley, 2002.

[15] B. Moro, "The full mont," *Union Bank of Switzerland, Published in RISK magazine*, 1995.

[16] M. P. Craig Kolb, *Chapter 45. Option pricing on the GPU, GPU Gems 2*. Addison-Wesley Professional, 2005.

[17] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 13–24.

[18] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a mapreduce framework on graphics processors," in *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. New York, NY, USA: ACM, 2008, pp. 260–269.

[19] "Hadoop streaming," *available at hadoop.apache.org*.

[20] R. S. Bird, "An introduction to the theory of lists," in *Proceedings of the NATO Advanced Study Institute on Logic of programming and calculi of discrete design*. New York, NY, USA: Springer-Verlag New York, Inc., 1987, pp. 5–42.

[21] M. Cole, "Parallel programming with list homomorphisms," *Parallel Processing Letters*, vol. 5, pp. 191–203, 1995.

[22] S. Gorlatch, "Constructing list homomorphisms," Universitt Passau, Germany, Tech. Rep., 1995.

[23] K. Kakehi, Z. Hu, and M. Takeichi, "List homomorphism with accumulation," in *In Proceedings of Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD*, 2003, pp. 250–259.

[24] A. Morihata, K. Matsuzaki, Z. Hu, and M. Takeichi, "The third homomorphism theorem on trees: downward & upward lead to divide-and-conquer," in *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 2009, pp. 177–185.

[25] R. Lämmel, "Google's mapreduce programming model — revisited," *Sci. Comput. Program.*, vol. 68, no. 3, pp. 208–237, 2007.

[26] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, "Merge: a programming model for heterogeneous multi-core systems," *SIGPLAN Not.*, vol. 43, no. 3, pp. 287–296, 2008.

[27] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "Gpu cluster for high performance computing," in *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2004, p. 47.

[28] Z. Fan, F. Qiu, and A. E. Kaufman, "Zippy: A framework for computation and visualization on a gpu cluster," *Computer Graphics Forum*, vol. 27, no. 2, pp. 341–350, Apr. 2008.

[29] T. Tu, C. A. Rendleman, D. W. Borhani, R. O. Dror, J. Gullingsrud, M. O. Jensen, J. L. Klepeis, P. Maragakis, P. Miller, K. A. Stafford, and D. E. Shaw, "A scalable parallel framework for analyzing terascale molecular dynamics simulation trajectories," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–12.

[30] J. C. Phillips, J. E. Stone, and K. Schulten, "Adapting a message-driven parallel application to gpu-accelerated clusters," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–9.