Task-D: a Task Based Programming Framework for Distributed System

Jiachen Xue, Chong Chen, Lin Ma, Teng Su, Chen Tian, Wenqin Zheng, Ziang Hu
Huawei US R&D Center
Santa Clara, USA
{*jiachen.xue, chong.chen2, lin.ma, suteng, chen.tian, zhengwenqin, ziang.hu*}*@huawei.com*

***Abstract*—We present Task-D, a task-based distributed programming framework. Traditionally, programming for distributed programs requires using either low-level MPI or high-level pattern based models such as Hadoop/Spark. Task based models are frequently and well used for multicore and heterogeneous environment rather than distributed. Our Task-D tries to bridge this gap by creating a higher-level abstraction than MPI, while providing more flexibility than Hadoop/Spark for task-based distributed programming.**

The Task-D framework alleviates programmers from considering the complexities involved in distributed programming. We provide a set of APIs that can be directly embedded into user code to enable the program to run in a distributed fashion across heterogeneous computing nodes. We also explore the design space and necessary features the runtime should support, including data communication among tasks, data sharing among programs, resource management, memory transfers, job scheduling, automatic workload balancing and fault tolerance, etc. A prototype system is realized as one implementation of Task-D. A distributed ALS algorithm is implemented using the Task-D APIs, and achieved significant performance gains compared to Spark based implementation. We conclude that task-based models can be well suitable to distributed programming. Our Task-D is not only able to improve the programmability for distributed environment, but also able to leverage the performance with effective runtime support.

I. Introduction

Task-D is a task based distributed programming framework that supports distributed heterogeneous hardware platforms. Task-D framework consists of a programming library, which provides simple API interfaces that users can embed into their programs to express parallelism, and a runtime system that should provide services such as resource management, job scheduling, workload balancing and fault tolerance to facilitate the process of creation, deployment and execution of a distributed program.

Task based parallel programming model has a number of advantages. First, it expresses and constructs parallel applications naturally using sequential semantics. As shown in Cilk [1], removal of Cilk keywords would change the parallel program back into its serial version. When it comes to building distributed high performance programs, the Message Passing Interface (MPI) remains the de-facto standard. It is straightforward to express data parallelism using the MPI interface where the same function logic would be applied to different portions of the input data. Task based parallel programming model not only can be easily applied to express data parallelism, but also can be used to describe more general scenarios where distinct function logic can be executed in parallel and each works on different input data. Second, the dynamic nature of task based programming model also helps deal with irregularities in distributed applications. The irregularity of computation mainly comes from two major sources: non-uniformity of underlying computing infrastructure and sparsity in data. The cluster may be consisted of physical or virtual machines with different configurations which results in different processing speed even for the same job. Uneven density in data deteriorates the problem by introducing non-uniformly distributed tasks across the clusters. As tasks can be created dynamically and scheduled independently, spreading tasks evenly across the cluster based on real-time load information helps to achieve workload balancing and improve overall job throughput.

Similar to MPI, Hadoop [2] and more recent Spark [3] framework are efficient in expressing single program multiple data (SPMD) type workloads. On the other hand, they have provided a simple programming interface as well as services such as scheduling, failure recovery and resource management, which simplifies the process of writing a distributed application. However, both Hadoop and Spark hide communication from the programmer and thus is not capable to deal with applications with complex communication patterns.

Writing a distributed application often requires the programmers to deal with all kinds of complexities, such as node failure, heterogeneous hardware architectures, etc. To alleviate the programmer from considering all these difficult aspects involved in distributed computing, Task-D run-time should be able to hide all the details from the programmers by providing resource management, job scheduling, automatic workload balancing and fault tolerance to the user application. Task-D run-time should also provide support to applications that utilize GPUs on distributed computing nodes.

The rest of the paper is organized as follows. In Section II, we introduce the design philosophy of the proposed Task-D framework as well as its functionality. Task-D APIs design

and run-time requirement are expatiated in Section III-A. We discuss our prototype system, application and results in Section IV. Section V presents the conclusion of the paper.

II. The Task-D Framework

Task-D framework is a combination of user library and run-time system aiming to provide programmers an integrated environment for writing and running distributed applications. In Task-D framework, each user program is represented as a job, and within a job, functions that can be executed in parallel are called tasks. Jobs can be submitted to Task-D run-time either using command line or through web interface. The Task-D run-time needs to monitor real time resource utilization of the cluster, and makes scheduling decisions based on cluster-wide load information wisely. The Task-D run-time should also perform failure recovery in a programmer oblivious way.

A. Task Model

In Task-D, task is defined as a function plus its input data, and it is the most basic unit of expressing parallelism. Tasks can be created dynamically, and the Task-D run-time scheduler should be responsible for deciding when and where to execute the task. Task-D run-time needs to maintain a hierarchical task graph to store task related meta data. Task graph is a Direct Acyclic Graph (DAG) with each node denoting a task and edge representing task completion dependencies. As shown in Figure 1, there exists unique path through the job root to each created task, which resembles the idea of file path used in Linux file system. Such a path can be used to locate any tasks existed in the system.

Our Task-D model has advantageous benefits in several aspects. First, it provides a higher level abstraction than MPI, thus more suitable for dynamic and complicated tasks. Also, Task-D provides more flexible control in handling complicated communication patterns and skew than Hadoop/Spark. Rather than hiding all the communication from programmers, Task-D exposes pattern-based communication and scheduling interfaces with well-defined APIs which will be introduced in Section III-A. Due to the flexible data communication among tasks and efficient data sharing among different programs, tasks can be controlled and formulated at an appropriate granularity that the trade-off between high cluster utilization from long-running batch tasks and high responsiveness of interactive jobs from obtaining resources quickly. In addition to that, Task-D also has lower launch overhead than the popular Hadoop/Spark, which need to launch a new JVM for each task in many seconds. By maintaining an active thread pool for each possible worker node, task launch overhead can be reduced to a level as to make a remote procedure call [4].

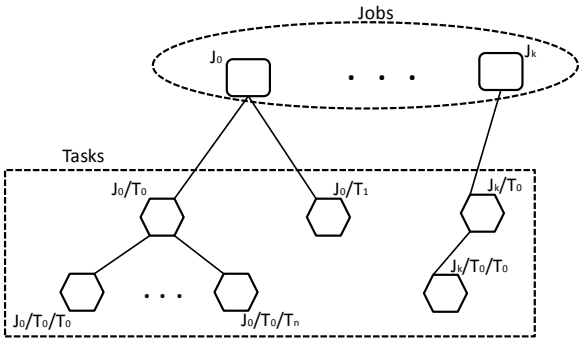

Figure 1: Sample task graph. Task-D run-time maintains task dependencies in a hierarchical manner.

B. Memory Model

Task-D run-time should map each task to an operating system process, which has its own memory address space. To communicate with other tasks, each task allocates dedicated memory region intended for communication, which is called communication buffer (***CB***), and also assigns a unique name to it within the task. Task-D library provides a set of one-sided communication routines for data exchange among different ***CB***s. Each ***CB*** has the same life span as the job that creates it. ***CB*** can be reclaimed either through explicit API call or automatically by run-time after the job is complete.

Task-D also provides a way to persist ***CB*** in memory unless deleted explicitly by the programmer. Such mechanism is benefits for the following reasons. First of all, it enables efficient in-memory data sharing among jobs, otherwise, data needs to be written back to the disk and shared in the form of files. Secondly, it provides an interface to third party data storage systems where useful data, not intermediate results, can be kept. Moreover, as we will show in Section IV, it simplifies the programming as such ***CB*** can be located solely by its name. Each of these ***CB***s should be given a unique name in the global sense, which serves as the key, since it is independent of the job once created, and referencing it by name is the most straight forward way.

III. Implementation

A. The Task-D Library API

In the following, we present a subset of APIs that are most common to users of Task-D. Task management and data communication are the two most important types of Task-D APIs.

1) Task Management:

- ***task (handle, attr, start_routine, args)***
- ***task_sync ()***

The ***task*** function call provides a straightforward way to pack a function and its input data, given by ***start_routine*** and ***args***

respectively, into a Task-D task. Additional task properties, such as priority, can be specified through ***attr***. On successful return, a task handle is passed back to ***handle***, which can be used later to reference this created task.

The caller of ***task*** is considered to be the parent, while the created tasks are called children. Sibling tasks are those sharing the same parent. A call to ***task_sync*** stalls the parent task till all of its children tasks are finished. Asynchronous data communication API could be used inside tasks. The return of ***task_sync*** does not guarantee data communication of its children tasks are complete. As a result, the programmer needs to use data synchronization API, provided in Section III-A3, to ensure proper data delivery.

2) Data Management:

- ***allocate (name, size, flag, scope)***
- ***delete (name)***

allocate creates a Communication Buffer (***CB***) of ***size*** with the given ***name***. ***flag*** specifies read/write permissions of the ***CB***, and ***scope*** takes one of the two values: DATA_LOCAL and DATA_PERSIST. As we discussed in Section II-B, ***CB*** created using DATA_LOCAL mode, which is the default scope value, would have the same life span as the job that creates it, while the DATA_PERSIST indicates that the ***CB*** would persist in memory unless being removed explicitly.

The ***delete*** API call frees up the memory region occupied by the ***CB*** , which is created by the current task.

3) Data Communication:

- ***get (buffer, source, name, offset, size)***
- ***set (buffer, dest, name, offset, size)***

The ***get*** API call reads ***size*** bytes of data from ***CB*** that can be located by <source, name>, starting at ***offset***, to local memory region specified by ***buffer***. ***set*** does the opposite; it writes the data stored in ***buffer*** to ***CB*** identified by <dest, name>. ***source/dest*** in get/set is a task handle needs to be passed into the task as a parameter. This may be unnecessary if the communication pattern is well defined, for example, when task wants to communicate with its sibling or parent. Therefore, we assign a rank to each task. A rank is a monotonically increasing non-negative integer starting from 0, which is assigned to the parent task. Task-D provides ***get/set*** routines taking rank, instead of task handles, as source/dest. To get/set values from/to a persisted ***CB***, one can simply pass NULL as the value of the source/dest.

Both ***get*** and ***set*** are blocking operations, and the non-blocking version is shown below.

- ***get_a (buffer, source, name, offset, size, request)***
- ***set_a (buffer, dest, name, offset, size, request)***

The extra ***request*** argument is a request handle passed back after each function call, which will be used in the ***data_wait*** function shown below to test if the non-blocking data communication operation is finished.

- ***data_wait (request)***

To provide synchronization among different data operations to the same ***CB***, user can surround data operation with a pair of lock and unlock API calls to ensure atomic updates. In the following, we show lock and unlock API for both local and persisted ***CB***. For protection of persisted ***CB***, the task parameter should be set as NULL.

- ***data_lock (task, name)***
- ***data_unlock (task, name)***

B. Runtime Design Requirement

1) Job and Task Scheduling: Before submitting a job to the Task-D run-time, user needs to provide information regarding the resource requirement of the program in terms of number of nodes, number of CPUs per node and amount of memory per node. The Task-D run-time should maintain a FIFO job queue, and submit the job to the cluster for execution when there is enough resource.

The tasks of a job can be created dynamically during the job execution. Task-D run-time needs to perform hierarchical work stealing [5] in an effort to achieve workload balancing. Each task keeps a record of its parent and the node ID of the parent task. In case the task being stolen by a remote node, after the task is complete, it is still able to notify its parent based on the parent task ID and its node ID. The parent task is considered to be complete when all its children tasks are finished. To locate a stealing target, it traverses upwards the task graph, seeks any nodes with unfinished tasks, and steals half of the tasks from it.

2) Fault Tolerance: By default, Task-D run-time is required to perform periodic coordinated checkpointing [6] for each job. The coordinated checkpointing requires to halt all current computation and communication to perform a global snapshot, and resume execution when the checkpointing is finished. The process of coordinated checkpointing is relative expensive and may not suitable for short execution jobs. Users of small jobs may opt not to use the default fault tolerance scheme, and the Task-D run-time will automatically resubmit the job to the job queue if the previous run of the job is not successfully completed.

IV. APPLICATION

In this section, we will demonstrate the simplicity and effectiveness of programming distributed application via using our Task-D framework on an algorithm called Alternating Least Squares (ALS), and run it on our prototype system.

A. ALS Algorithm

The ALS algorithm [7] has been widely used in big data analysis applications. The Spark framework also includes the ALS algorithm as a build-in function for its MLlib suite.

Consider that there is a data set which records the rating value of N users related to J items. One user will normally rate only a small fraction of the item set. It is not necessary for the user to evaluate all items. Because most of user-item pairs do not relate to a valid rating, the data set can be described by a sparse matrix R, while the value of known elements $R(i,j)$ represents the rating of the i^{th} user for the j^{th} item. The ALS algorithm is able to calculate the value of the missing elements of a sparse user-rating matrix based on the existing values of the matrix.

It effectively predicts the value of the missing elements of matrix R by factorizing it to two separate matrices: U and M. U and M are two dense matrices with dimension $N \times K$ and $K \times J$, where K is an adjustable parameter. In this paper, we denote K as the rank of ALS. The ALS algorithm iteratively minimizes the square error of the factorization by alternatively conducting equations (1),(2),(3) [7]. Consider the matrix M is fixed, the ALS algorithm will update the i^{th} row of matrix U using the following equations. As described in [7], n_{U_i} denotes the number of ratings of the user i and movie j that the user rated; E is the identity matrix; M_{I_i} denotes the sub-matrix of M where column j belonging to I_i are selected; $R(i, Ii)$ is the row vector where column j belonging to I_i of the i^{th} row of R is taken.

$$u_i = {A_i}^{-1} V_i \tag{1}$$

$$A_i = M_{I_i} M_{I_i}^T + \lambda n_{U_i} E \tag{2}$$

$$V_i = M_{I_i} R^T(i, I_i) \tag{3}$$

```
#define Rank
#define UserNum
#define MovieNum

struct ALS_arg {
  int tid;
  char type;
};

void *initALS(void argALS) {
  struct ALS_arg args;
  allocate(``R'', sizeR, D_READ|D_WRITE, DATA_PERSIST);
  allocate(``U'', sizeU, D_READ|D_WRITE, DATA_PERSIST);
  allocate(``M'', sizeM, D_READ|D_WRITE, DATA_PERSIST);

  task_t t[procN];
  for (int iter=0; iter<maxIter; iter++) {
    for (int i=0; i<procN; i++) {
      t[i] = task(&t[i], NULL, ALS, (void *)&args);
    }
    task_sync();
  }
}
```

Listing 1: ALS algorithm initialization using Task-D API

```

void ALS(void *args) {
  // ........
  // Omit data allocation and only demonstrate
  // the update of matrix M

  if (input->type=='M') {
    // Fix U, update M; Get data from global buffer
    startRow = tid*UserNum/ProcNum;
    if (tid==ProcNum-1) {
      endRow = UserNum;
    } else {
      endRow = (tid+1)*UserNum/ProcNum;
    }
    for (int i=startRow; i<endRow; i++) {
      for (int j=0; j<MovieNum; j++) {
        if (R(i,j)) {
          get(vector1, NULL, ``U'', j*Rank, Rank);
          update_vector(``vector2'', ``vector1'', R(i,j));
          update_matrix(``vector1'', ``matrix'');
        }
        solve(``matrix'', ``vector2'', ``vector3'');
      }
      set(vector3, NULL, ``M'', i*Rank,
          Rank*sizeof(double));
    }
  } else if (input->type=='U'){
    // Omit the update of matrix U, which is similar to
    // the procedure of updating matrix M
    // ........
}
```

Listing 2: ALS algorithm implemented using Task-D API

```
// ........
// Only present the procedure of updating matrix M

startRow = tid*UserNum/commSize;
if (tid == commSize-1) {
  endRow = UserNum;
} else {
  endRow = (tid+1)*UserNum/tid;
}

// MPI send/recv operation
for (int i=0; i<commSize; i++) {
  if (i==tid) {
    for (int j=0; j<commSize; j++) {
      if (j != tid) {
        MPI_Recv(recvBuffer[recvBufferOffset[j]],
             recvSize[j],
               MPI_DOUBLE, j, j, MPI_COMM_WORLD);
      }
    }
  } else {
    assemble(localU, &sendAssembleTable[boundary[i]],
          i, boundary, sendBuffer);
    MPI_Send(sendBuffer, sendSize[i], MPI_DOUBLE, i, i,
         MPI_COMM_WORLD);
  }
  MPI_Barrier(MPI_COMM_WORLD);
  // Loop for updating
  for (int i=startRow; i<endRow; i++) {
    for (int j=0; j<MovieNum; j++) {
      if (R[i][j]) {
        // Get the rank of nodes
        jRank=getRank(j);
        if (jRank != tid) {
          fetch(vector1, j,
               &recvBuffer[bufferIndex[jRank]]);
        }
        updateVector(vector2, vector1, R[i][j]);
        updateMatrix(vector1, matrix);
      }
```

```
      // Solve the linear system to get the update for
      // matrix M in the ith row
      solve(matrix, vector2, vector3);
      for (int k=0; k<rank; k++) {
        localM[j*rank+k] = vector3[i];
      }
    }
  }
}
```

Listing 3: The MPI based implementation of ALS algorithm

The code of Task-D based ALS is demonstrated in Listing 2. The Task-D function $ALS()$ is distributed to multiple computing units to derive the factorization of the sparse matrix R. The HARE-D run-time executes the program and alternatively updates matrix U and M. The function $updateVector()$ and $solver()$ implement the equation (1) and (2), and not expatiated due to their irrelevance to the programming model we proposed in this paper.

Listing 3 demonstrated a MPI based implementation of matrix updating for the ALS algorithm. We opted not to use one-side communication APIs such as MPI_Put() and MPI_Get() because the one-sided communication routines in MPI require to place MPI_Win_fence() after every vector read, which is not suitable for ALS because it is highly possible that multiple computer nodes will simultaneously read vectors from the same memory window. MPI share the data directly using the MPI_Send and MPI_Recv. When implementing the distributed computing using MPI, there is no method to allocate a globally shared memory space. As demonstrated in Listing 3, to efficiently transfer data between computer nodes, we should carefully apply the data transfer API to explicitly transfer the specified amount of data to certain computer nodes of the system. The MPI API only deals with continuous memory space, which causes a problem in our implementation because the shared data is not stored continuously in memory. We had to implement an assemble() function to assemble discrete data into an array. We also implemented a fetch() function to fetch required data from recvBuffer. Compared with the MPI based implementation, it is obvious that the Task-D framework greatly simplifies the data communication involved in user programming.

B. Prototype System and Experimental Results

We developed a prototype system as an implementation of Task-D framework for distributed computing development. Task-D serves as the programming model for developing applications with back-end run-time support. We uses redis as the distributed file system for data share between computing nodes and distributed memory system. The redis operations are integrated into the Task-D APIs. We adopt a coarse grained task sharing scheme to enable tasks being executed among different computing nodes.

We examine the Task-D based ALS implementation on a multi-core machine equipped with a 4-core (8 logic cores) i7-4770k CPU and 8 GB memory, and compare with a Spark implementation on the same machine. We assume one logic core as an independent computing node. By using the Task-D APIs, all the communications between cores are achieved through redis (i.e. we do not use shared memory to share the intermediate result between logic cores). The input data is the movilens 1 million rating data. Eight tasks are initialized to compute the recommendations for the movielen data, each assigned on a single core. To explore the complexity of the problem, the rank of ALS is increased from 8 to 100. The experiment results with 10 iterations are demonstrated in Table I. This shows that for a variety of rank numbers (problem sizes), the Task-D implementation achieves similar root-mean-square error (RMSE), but up to 4 times faster than our spark implementation on the same platform.

Rank Number	Execution Time (s)	RMSE
8	4.9	0.801560
32	51	0.718798
64	180	0.674684
100	434	0.652417

Table I: Execution time and the RMSE of the implementation of ALS algorithm with Task-D runtime.

In Figure 2, we show how Task-D runtime scales with the number of processes. For strong scalability test, we increase the number of processes from 2 to 32 while fix the rank of the problem to 128. By increasing the number of processes from 2 to 32, we are able to get 13x speedup. The complexity of the ALS algorithm increases quadratically with the rank of the problem. For weak scalability test, with 2 processes, we have a problem of rank 128. For 8 processes, we used a problem of rank 256, and a problem of rank 512 is supplied to the case where 32 processes are used. As shown in Figure 2, the time consumed for each case remains stable with the increase of both problem complexity and number of processes.

V. Conclusion

In this paper, we present a task based distributed programming framework composed of a complete set of APIs and runtime requirement, named Task-D. It has an abstraction layer sitting between MPI and Hadoop/Spark, and flexibly handles efficient data sharing, communication, resource management, job scheduling, workload balancing, and fault tolerance. We demonstrate the simplicity and effectiveness of programming distributed applications by applying the Task-D framework to a big data analysis algorithm called ALS running on our implemented prototype system. Our results

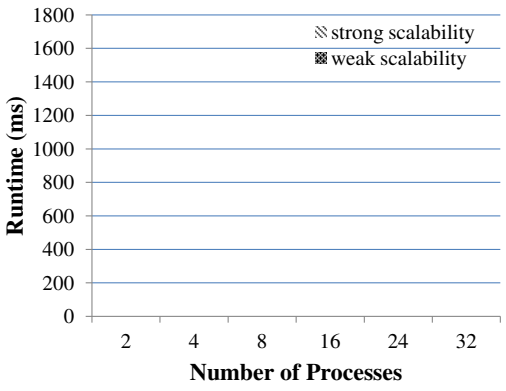

Figure 2: Scalability test of Task-D with ALS algorithm. The rank of the problem is set to 128 for strong scalability test. For weak scalability test, we double the problem rank while quadruple the number of processes.

indicate that programmability using Task-D is improved, and with similar levels of prediction errors, the Task-D results outperform the Spark results by up to 4-fold. Task-based models like Task-D can be effectively used for distributed programming environment.

In future work, we plan to extend Task-D runtime to utilize accelerators, such as GPUs [8], [9], [10] and FPGAs. To explore the full potential of the heterogeneous system requires detailed modeling [11], [12] of the interaction between the accelerators and the host. We plan to apply Task-D runtime to a larger scope of applications such as high performance computing [13] and machine learning [14]. We also plan to design a dynamic resource scheduling subsystem [15] to facilitate the deployment of Task-D runtime in cloud environment.

REFERENCES

[1] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, ser. PLDI '98. New York, NY, USA: ACM, 1998, pp. 212–223.

[2] T. White, *Hadoop: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2009.

[3] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 2–2.

[4] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica, "The case for tiny tasks in compute clusters," in *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, ser. HotOS'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 14–14.

[5] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 53:1–53:11.

[6] Y. Tamir and C. H. Squin, "Error recovery in multicomputers using global checkpoints," in *In 1984 International Conference on Parallel Processing*, 1984, pp. 32–41.

[7] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan, "Large-scale parallel collaborative filtering for the netflix prize," in *Proceedings of the 4th International Conference on Algorithmic Aspects in Information and Management*, ser. AAIM '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–348.

[8] L. Ma, K. Agrawal, and R. D. Chamberlain, "A memory access model for highly-threaded many-core architectures," *Future Generation Computer Systems*, vol. 30, pp. 202–215, January 2014.

[9] L. Ma, K. Agrawal, and R. D. Chamberlain, "Analysis of classic algorithms on GPUs," in *Proc. of the 12th ACM/IEEE Int'l Conf. on High Performance Computing and Simulation (HPCS)*, 2014.

[10] L. Ma, K. Agrawal, and R. D. Chamberlain, "Theoretical analysis of classic algorithms on highly-threaded many-core GPUs," in *Proc. of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2014, pp. 391–392.

[11] L. Ma and R. D. Chamberlain, "A performance model for memory bandwidth constrained applications on graphics engines," in *Proc. of Int'l Conf. on Application-specific Systems, Architectures and Processors (ASAP)*, 2012.

[12] L. Ma, R. Chamberlain, and K. Agrawal, "Performance modeling for highly-threaded many-core GPUs," in *Proc. of Int'l Conf. on Application-specific Systems, Architectures and Processors (ASAP)*, June 2014, pp. 84–91.

[13] C. Chen and T. M. Taha, "A communication reduction approach to iteratively solve large sparse linear systems on a gpgpu cluster," *Cluster Computing*, vol. 17, no. 2, pp. 327–337, Jun. 2014. [Online]. Available: http://dx.doi.org/10.1007/s10586-013-0279-2

[14] L. Ma, R. D. Chamberlain, J. D. Buhler, and M. A. Franklin, "Bloom filter performance on graphics engines," in *Proc. of Int'l Conf. on Parallel Processing*, 2011, pp. 522–531.

[15] Y.-J. Hong, J. Xue, and M. Thottethodi, "Selective commitment and selective margin: Techniques to minimize cost in an iaas cloud," in *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems & Software*, ser. ISPASS '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 99–109. [Online]. Available: http://dx.doi.org/10.1109/ISPASS.2012.6189210