

# A Programming Model for Massive Data Parallelism with Data Dependencies \*

Yongpeng Zhang, Frank Mueller,  
North Carolina State University  
Raleigh, NC 27695-7534  
yzhang25@ncsu.edu, mueller@cs.ncsu.edu

Xiaohui Cui, Thomas Potok  
Oak Ridge National Laboratory  
Oak Ridge, TN 37831  
cuix@ornl.gov

## Abstract

*Accelerating processors can often be more cost and energy effective for a wide range of data-parallel computing problems than general-purpose processors. For graphics processor units (GPUs), this is particularly the case when program development is aided by environments, such as NVIDIA's Compute Unified Device Architecture (CUDA), which dramatically reduces the gap between domain-specific architectures and general-purpose programming. Nonetheless, general-purpose GPU (GPGPU) programming remains subject to several restrictions. Most significantly, the separation of host (CPU) and accelerator (GPU) address spaces requires explicit management of GPU memory resources, especially for massive data parallelism that well exceeds the memory capacity of GPUs.*

*One solution to this problem is to transfer data between GPU and host memories frequently. In this work, we investigate another approach. We run massively data-parallel applications on GPU clusters. We further propose a programming model for massive data parallelism with data dependencies for this scenario. Experience from micro benchmarks and real-world applications shows that our model provides not only ease of programming but also significant performance gains.*

## 1 Introduction

Data-parallel coprocessors provide attractive accelerating platforms for the growing market of parallel computing. Not only do they usually offer better raw performance compared to CPU-only programs, but also have better cost-

performance and energy-performance ratios. The wide usage of data-parallel coprocessors, such as GPUs in the general computing domain, is boosted by the CUDA [3] programming model by NVIDIA and OpenCL (Open Computing Language) standard [1] by Apple. CUDA, by introducing a few extensions to C as a programming interface to GPUs, lowers the barrier for general application programmers to program on GPUs. CUDA predominantly runs on NVIDIA's GPUs but has also been ported to general-purpose multicores [15]. OpenCL, an open industry standard, tries to provide a vendor-independent programming model for all processors not restricted to GPUs.

Even though it is much easier to program on GPUs than ever, certain restrictions still apply. For instance, GPUs do not support virtual addressing. Thus, the data size they work on at one time is limited by the size of the on-board physical memory. As the problem size increases, it is necessary to explicitly transfer the data in and out of GPU memory frequently. Delicate balance between the computation and communication is often required to achieve the best performance. This approach is error-prone and may be sensitive to hardware upgrades because the balance may be broken with a different generation of hardware supporting more or less memory space.

With GPGPU programming, clusters of nodes equipped with coprocessors seem to be a promising solution for increasingly large problems. In homogeneous cluster computing where no accelerators exist in the system, the message passing via the Message Passing Interface (MPI)[2] is the dominant programming model. Therefore, it may be tempting to incorporate MPI directly to support GPU clusters. However, since MPI is tailored for CPU clusters in the first place, new problems arise:

- There is no direct link between a coprocessor's memory and the network interface. To achieve cross-node message passing, system memory is required as a bridge between the coprocessor and the network. This overhead does not exist in homogeneous systems and adds extra programming efforts to realize message

---

\*Copyright Notice: This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes.

passing between memory of different GPUs.

- One of the many lessons learned to obtain good speedup in single GPU machine is to keep the GPU as busy as possible. With a new level of memory hierarchy introduced, it becomes even harder to fully utilize the GPU's computational resources in clusters than that in single GPU systems if only one MPI process takes full control of one GPU.

To solve these problems, we propose a framework that is layered on top of MPI to supply both ease of programming and performance. Its key characteristics are:

- It provides the means to spawn a flexible number of threads for a parallel task without being restricted to the number of nodes in the system. Multiple threads can run in the same MPI process and share one GPU card to increase the utilization rate. In applications where the communication vs. computation ratio is relatively high, this is an important optimization option.
- A distributed object allocation interface is introduced to merge CUDA memory management and explicit message passing commands. This is based on the observation that the management of the on-board physical memory consumes a large amount of coding effort for the programmer. The need to exchange data through MPI even complicates matters further. Our interface helps programmers view the application from a data-centric's perspective. From the performance point of view, the underlying run-time system can detect data sharing within the same GPU. Therefore, network pressure is reduced, which is becoming more important because of the computation time reductions as coprocessors increase in speed.
- An interface for advanced users to influence thread scheduling in clusters is provided, motivated by the fact that the mapping of multiple threads to physical nodes affects the performance depending on the application's communication patterns.

## 2 Programming Model

The computational platform we consider here is a cluster of closely coupled machines connected via a fast network link (Figure 1). There are three levels of parallelisms. At the top level are the MPI processes, the number of which is determined by the availability of resources, e.g., the number of physical nodes in the cluster. At the middle layer are the CPU threads that run in each MPI process. They share one GPU card and launch GPU kernels independently. At the bottom level are the GPU kernels consisting of blocks of many light-weighted threads.

The user writes MPI-like programs that will later be automatically pre-processed into native MPI programs running on clusters. A program is composed of serial parts and data-parallel parts. For the serial part of the program, every node executes the same code path. The data parallel part is indicated by specifying the function name, dimension and arguments. Implicit barriers exist at the end of each data-parallel part. From now on, we will refer to the functions that are executed in the parallel region as *kernels*. Our model does not have the constraint of having less (equal) number of tasks than (to) the number of nodes available in cluster, as in the MPI program. In fact, it may be advantageous to spawn more tasks than the cluster size to achieve better load balancing.

### 2.1 Distributed Object Interface

Even though tasks are running on different nodes in the data-parallel part of the program, they can still communicate with each other through a *Distributed Object Interface*. Every data object that exists and shares data across the computational nodes can be registered in the system as a *distributed object*. Currently, the model supports 1-D, 2-D and 3-D objects of any type. The interface defines the following operations:

- *RegDObj(name, [dimension], [unitbytesize])*: The declaration of a *distributed object*. This method is called in the serial part of the program. Only the name of the *distributed object* is needed at declaration time. The provided object name will be used to refer to this object in other *distributed object interfaces*. The dimension and unit size of this object are optional at the declaration point to provide the maximal flexibility.
- *pair<void \*, devicePtr \*> GetDObj(name, range, unitbytesize)*: This method is called in the task of the data-parallel part and returns two pointers: one to the system memory, the other to the GPU's device memory. A task can obtain a copy of a memory slot of the *distributed object*. It is transparent to the programmer where this data originated from. The existence of this copy on this node is *not* made public to the other node at this point. Both blocking and non-blocking versions of this method are provided to give the user the flexibility to overlap computation and communication.
- *ModifiedDObj(name, range, unitbytesize)*: This method is called in the task of the data-parallel part. The task claims to exclusively own the copy of a memory slot. The result of this operation will free spaces of all other copies in other nodes that it had previously obtained a copy of. It also implies that later requests to the same memory slot will refresh data from this node.

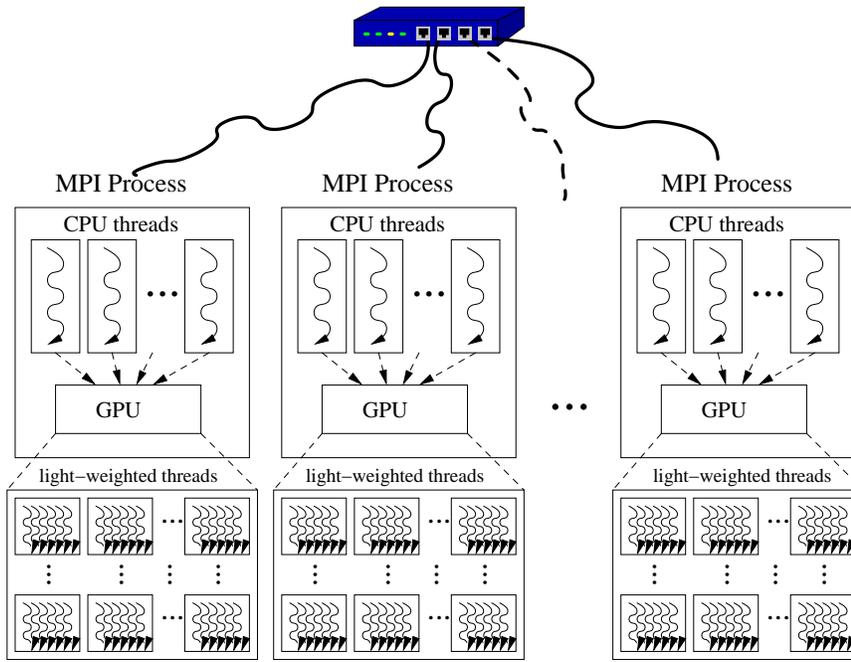


Figure 1. Execution Model

## 2.2 Case Study – 2D Stencil

We take a 5-point 2D stencil computation as an example to explain the usage of the *distributed object interface*. The 2D stencil computation is an iterative algorithm that, given data with a 2D dimensional layout, repeatedly updates the data by weighting neighboring data from the previous iteration with constant weighting factors:

$$D_{i,j}^{k+1} = w_1 * D_{i-1,j}^k + w_2 * D_{i+1,j}^k + w_3 * D_{i,j-1}^k + w_4 * D_{i,j+1}^k + w_5 * D_{i,j}^k$$

where  $D_{i,j}^k$  stands for the data at position (i,j) at iteration k and  $w_*$  are constant weighting parameters.

The pseudo-code for the main entry is shown below. Given a 2D dimension of  $N \times N$  as the total problem size and  $G \times G$  as the granularity of the parallel tasks (line 6), we can divide the entire job into  $S \times S$  tasks, each of which operates on a tile of the 2D plane. The data that represents the plane is declared as a *distributed object* in the system. To avoid synchronization issues within the kernel, we create two objects (“Plane1” and “Plane2”) representing the same plane. We then switch the input and the output for consecutive iterations (line 11, 12).

To specify that the boundaries of this tile will be fetched by neighbor tiles, the four margins are explicitly registered. For similar reasons, the region buffers are registered twice (line 14-15). If this step was skipped, the neighbor task

could just request a memory slot of the “Plane” object directly and the run-time system could provide the requested region on-the-fly. But the explicit boundary registrations are to reduce the run-time overhead by conserving memory usage. We can also see that  $S$  is used to specify the kernel dimension (line 19).

Listing 1. 2D Stencil Main Entry Pseudo-code

```

1 int main(int argc, char **argv)
2 {
3     // Serial region
4     // dimension of the plane is N by N
5     // plane is split by G by G tiles
6     int N, G;
7
8     // No. of tasks in each dimension
9     int S = (N-1)/G + 1;
10
11     RegDDData("Plane1", Dim(N, N, 1));
12     RegDDData("Plane2", Dim(N, N, 1));
13
14     RegDDData("North1", Dim(S, S, 1));
15     RegDDData("North2", Dim(S, S, 1));
16     // similar code for South, East and West
17     ...
18     for (int i = 0; i != MAX_ITER; i++)
19         LaunchKernel(Dim(S,S,1), N, G, i); // parallel region
20 }

```

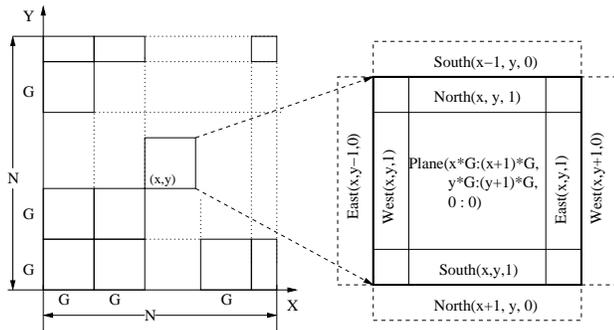
The stencil iteration is done in the following data-parallel function that is executed in parallel in the cluster. This function has a built-in variable called “gridIdx” of type *Dim* to facilitate the specification of the memory slot of the data this task is working on (line 3-6). Figure 2 gives a close-up view of the tile that a task at the gridIdx of  $(x, y, 1)$  operates on. This task will update data on the “Plane” object in the range from  $(x * G, y * G, 1)$  to  $((x + 1) * G, (y + 1) * G, 1)$ ; and four boundary objects in the range from  $(x, y, 1)$  to  $(x, y, 1)$ , inclusively. It will also need to fetch the data from the neighboring four tiles with shifted ranges.

**Listing 2.** 2D Stencil Thread Pseudo-code

```

void Stencil2D(...)
{
  1
  2
  3 int globalx = gridIdx.x * G;
  4 int globaly = gridIdx.y * G;
  5 int sizex = min((gridIdx.x + 1) * G, N) - globalx;
  6 int sizey = min((gridIdx.y + 1) * G, N) - globaly;
  7
  8 // Determines which buffers are inputs or outputs:
  9 // buffer 1s are the inputs, buffer 2s are the outputs
 10 <h_inplane, d_inplane> = GetDObj("Plane1",
 11     (gridIdx.x * G : gridIdx.x * G + sizex - 1,
 12     gridIdx.y * G : gridIdx.y * G + sizey - 1, 0:0),
 13     sizeof(float));
 14 <h_outplane, d_outplane> = GetDObj("Plane2",
 15     (gridIdx.x * G : gridIdx.x * G + sizex - 1,
 16     gridIdx.y * G : gridIdx.y * G + sizey - 1, 0:0),
 17     sizeof(float));
 18
 19 // margin conditions are omitted here
 20 <h_mynorth, d_mynorth> = GetDObj("North1",
 21     (gridIdx.x : gridIdx.x,
 22     gridIdx.y : gridIdx.y, 0:0), sizeof(float) * sizey);
 23 //...similar code for mywest, mysouth and myeast

```



**Figure 2.** 2D Stencil Distributed Data View

```

24 //get neighbor margins as input
25 // (margin conditions are omitted)
26 <h_north, n_north> = GetDObj("South1",
27     (gridIdx.x - 1 : gridIdx.x - 1,
28     gridIdx.y : gridIdx.y, 0:0), sizeof(float) * sizey);
29 //... similar code for west, south and east
30
31 // the stencil core function
32 stencil_core(d_outplane, h_outplane, d_inplane,
33     h_inplane, d_north, ..., sizex, sizey);
34
35 // programmer needs to update the four margins explicitly
36 update_margins(<d_outplane, h_outplane>,
37     <d_mynorth, h_mynorth>, ..., sizex, sizey);
38
39 // claims exclusive ownership of the output object
40 ModifiedDObj(<h_outplane, d_outplane>, "Plane2",
41     (gridIdx.x * G : gridIdx.x * G + sizex - 1,
42     gridIdx.y * G : gridIdx.y * G + sizey - 1, 0:0),
43     sizeof(float));
44
45 ModifiedDObj(d_mynorth, h_mynorth, "North2",
46     (gridIdx.x : gridIdx.x, gridIdx.y : gridIdx.y, 0:0),
47     sizeof(float) * sizey);
48 //... similar code for mywest, myeast and mysouth
49 }
50

```

## 2.3 Global Synchronization

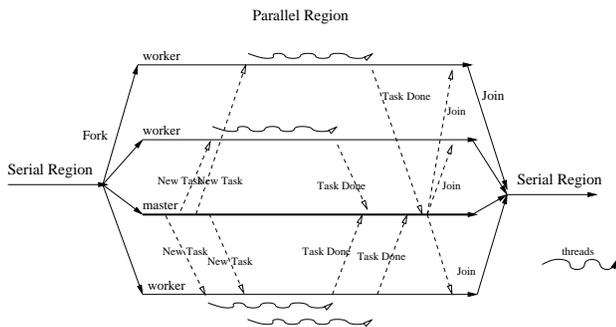
Our programming model provides a global barrier synchronization syntax `__syncthreads()` for parallel tasks running in the same parallel region. Tasks will block on this function until all other tasks have reached this function. We could add this barrier in the `Stencil2D()` function above just before the stencil core function (around line 31) to remove the double buffers for all *distributed objects* in the application. However, the message passing overhead involved with the synchronization would be significantly larger without double buffering for this stencil example.

## 3 Implementation

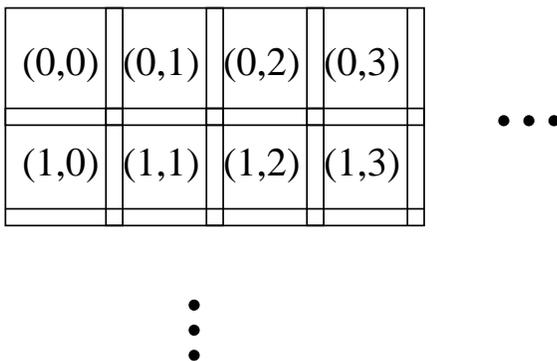
This section describes the implementation of the programming model and its supporting run-time system. The final executable is a native MPI program. We assume the number of physical nodes is equivalent to the number of processes, i.e., each node has one MPI process during the run-time.

### 3.1 The Foundations

The parallel execution uses fork/join style parallelism. One of the processes is chosen as a master process (usually but not necessarily rank 0) while other processes are defined as workers. For serial regions, master and workers execute the same serial code. Serial code can only involve global variables shared in all processes and does not have branches that affect the parallel region path of each process. For parallel regions of the program, the master launches a scheduler to assign tasks to workers according to the parallel region's task dimension. The number of tasks is purely application dependent and is specified by the kernel dimension shown in line 19 of Listing 1. All communication between master and workers is performed by MPI messages. Workers keep receiving message commands from the master to spawn a new thread for each task. After a task has completed, the associate worker informs the master. After all tasks are finished, the master sends *JOIN* messages to all workers to allow them to return to the serial region of the program (implicit global barrier). This parallel execution is illustrated in Figure 3.



**Figure 3. Parallel Execution between Master and Workers**



**Figure 4. Data Locality Aware Scheduling**

The new task command in the message from master to workers contains the task id and task dimension of the kernel. The kernel functions can use these parameters to select different regions of a shared data object, as is done in the `stencil2D()` function above (line 3-6).

### 3.2 Distributed Object

To support the *distributed object interface*, the master functions as a centralized database that keeps track of all regions of the *distributed objects* claimed by all previous `ModifiedDObj()` calls by the worker. If there is already an overlapping entry registered with the master when it receives a `ModifiedDObj()` request, it sends invalidation messages to the previous owner process to free the portion of data.

All workers also keep a local table that stores all exclusive copies of *distributed objects* it exclusively claims the ownership of. There are three cases when a task (task  $i$ ) request a piece of *distributed object* by `GetDObj()`:

1. The local table contains this entry. Then the run-time system just returns the pointer. No network messages are needed.
2. The local table does not contain this entry. The run-time system sends a request message to master. The master searches its global table. If no previous registrations are found, it will send a *INIT* message to the source indicating that the source can go ahead allocate the memory locally. No other node has the fresh data. Please note here that neither the master nor the worker will add the entry in their tables yet. The insertion should only be done when a worker issues a `ModifiedDObj()` call. Two network messages are required in this case.
3. If the master has found a match in another worker ( $j$ ), it will send a message back to worker  $i$  pointing to  $j$ . Worker  $i$  will then send another message to  $j$  directly request the data. The requested data will then be directly transferred from worker  $j$  to worker  $i$ . Four control messages and one data payload message are required in this case.
4. To reduce the number of control messages for non-local data fetching, a cache of source records is preserved in each worker. At the beginning of step 2, a worker can first consult the cache to see if previous requests have been replied. If that is the case, it can directly request the data from the source without consulting the master.

### 3.3 Scheduling with Data Locality

In programs running in clusters connected by a network, data locality is far more important than for those running on single node. This is because the network transfer speed is usually orders of magnitude slower than that through the internal memory bus. Therefore, performance is very sensitive to the run-time task mapping to the computation node. Take the 2D stencil as an example (Figure 4). Suppose there are three workers. An inefficient schedule would be to assign blocks to workers in a round-robin fashion, i.e., block(0,0) to worker1, block(0,1) to worker2 and block(0,2) to worker3. In this case, the exchange of data is maximized over the network. A better way of scheduling is to assign neighboring blocks (0,0) and (0,1) to worker1, blocks (0,2) and (0,3) to worker2, and blocks (1,0) and (1,1) to worker3. This way, some of the messages can be converted into internal memory copies.

We believe scheduling should be part of the supporting run-time system, too. But we also insist that an interface should be provided to advanced programmer to influence the scheduling decisions because the best scheduling may be application dependent. Our solution is to allow the user to specify the data requirement for each thread using a pragma-style specification. This specification will be turned into functions that are executed by the master scheduler to find the best mappings. The following criteria (in decreasing order of priority) are:

- Cache: for repeated calls to the same kernel, the previous mapping is applied (timing locality);
- Load balancing: workers will receive roughly the same number of tasks;
- Locality sensitivity: For a task that requires a certain data region as input, the worker with the larger intersection of data is chosen to execute the task;
- Initial decision: in the initial state, scheduler will temporarily store the data location information for tasks that have been scheduled and use this information to assign subsequent tasks based on the intersection metric.

## 4 Tool Chain and Software Stack

The software stack is composed of several layers as illustrated in Figure 5. MPI, the boost thread library and CUDA are used to support the three top-down levels of parallelism, respectively. Our framework combines the above three libraries to provide completely transparent multi-threading inside MPI process. The MPI and CUDA interfaces are partially exposed to the application layer. MPI calls can still be

used in the serial parts of the code and users are required to launch GPU kernels adhering to the CUDA driver API. Though advanced users can directly work on the framework API, an optional pre-processor is provided to reduce the lines of code.

It is worth noting that this framework can adjust to clusters without any CUDA-enabled GPUs. In this case, this framework transforms to a multi-threaded MPI programming model. The device memory pointers in the *Distributed Object Interface* are simply nullified. In that case, host memory pointers are directly passed to high-level language functions that replace the CUDA kernels.

Figure 6 shows the tool chain of the framework. The boost library needs to be compiled with the MPI enabled option. With code auto-generation, the pre-processor converts the spec-like user inputs to C++ codes, which are further compiled by the MPI compiler into MPI executable file. The final execution is performed in a typical MPI scenario.

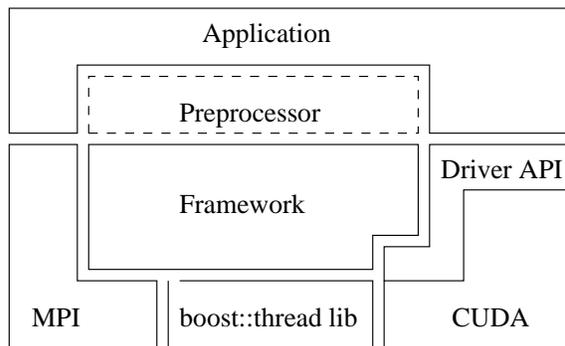


Figure 5. Software Stack

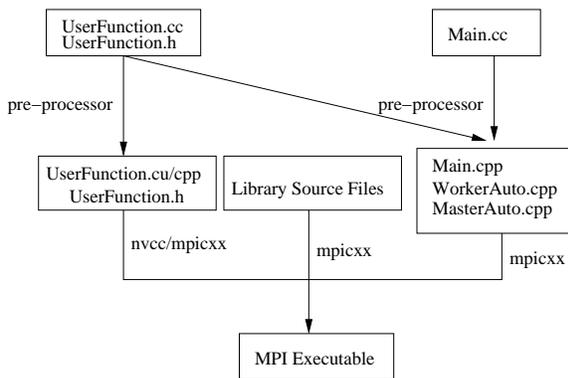


Figure 6. Tool Chain

## 5 Experimental Results

We have prototyped and profiled the run-time system in a cluster of up to sixteen nodes, each of which contains

a Geforce GTX 280 graphic card and a dual-core AMD Athlon 2 GHz CPU with 2GB of memory. For comparison purposes, execution time is measured for both the functionally equivalent GPU and CPU test benchmarks.

### 5.1 2D Stencil

The 2D stencil code performs calculations in manner reflecting computations of common particle codes in physics where a local value depends on that of their immediate neighbors in a multi-dimensional space represented as matrices. Figure 5.1 shows the execution time with various configurations over different problem sizes  $N$  in a 5-point stencil pattern. The single-CPU single-thread curve follows the  $N^2$  trend (only  $N \leq 10000$  are shown). Adding both CPU and GPU resources prove to be beneficial for  $N \geq 4000$ . It can be seen that the more computational resources we have in the system, the larger problem sizes we can handle. Overall, the GPU version provides a 6X speedup over the CPU version with the same number of nodes.

In Figure 8, we take a closer look at the communication and computation time ratios for multi-GPU and multi-CPU cases. All curves start from very high ratio for  $N < 5000$ . Since the communication time is roughly the same for both the CPU and GPU versions under the same number of nodes, the ratio between the GPU curve and the CPU curve implies the pure kernel acceleration rate, which is about 10X for larger  $N$ s.

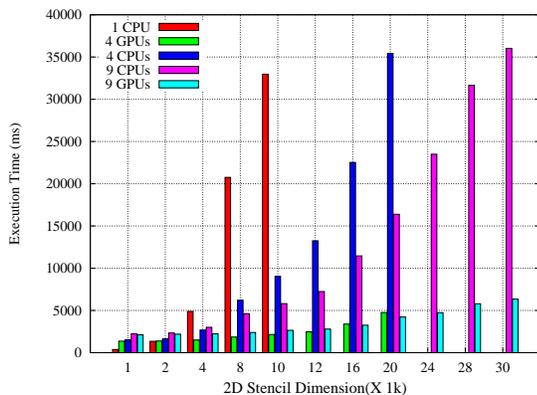


Figure 7. 2D Stencil Execution Time

### 5.2 Document Clustering

We assess this programming model using a real-world application, namely a massive document clustering problem. The complete algorithm is based on previous research performed at ORNL ([13, 4]) and consists of two phases:

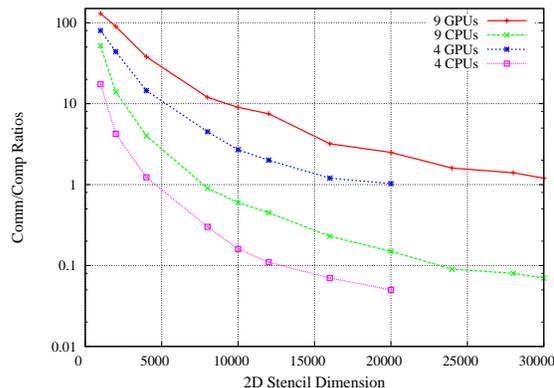


Figure 8. 2D Stencil Comm/Comp Ratios

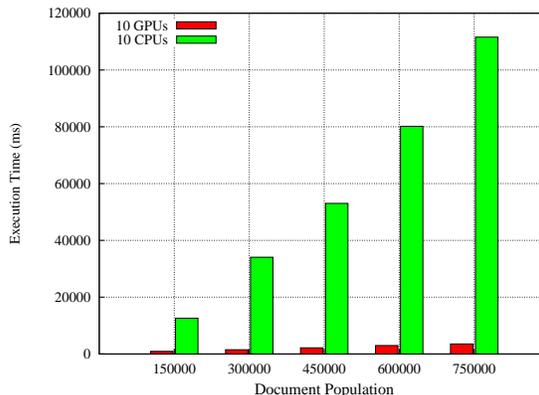


Figure 9. Tf-Icf Calculation

the Tf-Icf step and flocking-based document clustering simulation.

In the Tf-Icf step, the input is a corpus of plain-text documents. The goal is to reduce each document to a vector of unique terms associated with Tf-Icf values. This vector represents the characteristics of this document and can be used to calculate similarities of two documents given their term vectors.

The clustering simulation step maps one document as a moving point and simulates its positions in a bounded 2D plane. A similarity metric between neighboring documents is then calculated on-the-fly to determine each point's behavior (velocity) in each iteration. With enough computational nodes in the cluster, we are able to store all document term vectors in memory and transfer them as necessary with message passing.

The execution time in ten GPUs and CPUs for the Tf-Icf step is shown in Figure 9. The GPU version achieves approximately 30X speedup over the CPU version and has

made the disk I/O the predominant bottleneck in this step. A similar speedup is observed in the document clustering step.

## 6 Related Work

A myriad of work with respectable results has been reported in stand-alone GPU systems [9, 12, 14, 5, 10]. As more and more applications report speedups by data-parallel co-processing, the community is beginning to investigate the potential of massively data-intensive applications, such as text mining, clustering and classifications [18, 8, 17]. In most cases, the on-board physical memory is one of the major constraints to achieve scalability for industrial usage. In contrast, few experiments on GPU clusters can be found in literature. Two ad-hoc approaches including acceleration of scientific computations in GPU clusters [7, 6].

Recent work [16] proposes an MPI-like message passing framework, *DCGN*, for data-parallel architectures. This enables programmers to initiate message passing from inside the GPU kernel. The data synchronization for communication messages between device memory and system memory is transparently performed in the framework with only little performance overhead. In contrast, our work focuses on a coarser grained level in the sense that message passing is performed outside the GPU kernels.

Our programming model employs a hybrid parallelization model at the MPI process level, where parallel executions are forked and joined implicitly. This approach differs from previous work [11] that compiles existing OpenMP programs to CUDA kernel codes running in single node.

## 7 Conclusion

In this paper, we presented a novel programming model for massive data parallelism with data dependencies, particularly aiming to provide both performance and ease of programming for the emerging GPGPU clusters. A *Distributed Object Interface* was proposed to force programmers to think and design algorithms in a data-centric manner. Tedious and error-prone issues, such as thread mapping, scheduling and device memory management, are completely handled by the underlying framework to help programmers concentrate on application-specific design issues. Experimental results from a micro benchmark and a real-world application have shown dramatic speedups over traditional CPU clusters, ranging from 10X to over 30X. This demonstrates the strong performance potential of utilizing GPU clusters for large-scale applications.

## References

- [1] <http://www.khronos.org/opencv/>.
- [2] <http://www.mcs.anl.gov/research/projects/mpi/>.
- [3] [http://www.nvidia.com/object/cuda\\_home.html](http://www.nvidia.com/object/cuda_home.html).
- [4] X. Cui, J. Gao, and T. E. Potok. A flocking based algorithm for document clustering analysis. *J. Syst. Archit.*, 52(8):505–515, 2006.
- [5] M. Curry, L. Ward, T. Skjellum, and R. Brightwell. Accelerating reed-solomon coding in raid systems with gpus. In *International Parallel and Distributed Processing Symposium*, Apr. 2008.
- [6] E. Elsen, M. Houston, V. Vishal, E. Darve, P. Hanrahan, and V. Pande. N-body simulation on GPUs. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 188, New York, NY, USA, 2006. ACM.
- [7] 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*, page 47, Washington, DC, USA, 2004. IEEE Computer Society.
- [8] W. Fang, K. K. Lau, M. Lu, and X. Xiao. Parallel data mining on graphics processors. Technical Report HKUST-CS08-07, The Hong Kong University of Science and Technology, Oct 2008.
- [9] M. Fatica and W.-K. Jeong. Accelerating MATLAB with CUDA. In *HPEC*, Sept. 2007.
- [10] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *HiPC*, pages 197–208, 2007.
- [11] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 101–110, New York, NY, USA, 2009. ACM.
- [12] H. Nguyen(ed). *GPU Gems 3*. Addison-Wesley Professional, 2007.
- [13] J. W. Reed, Y. Jiao, T. E. Potok, B. A. Klump, M. T. Elmore, and A. R. Hurson. Tf-icf: A new term weighting scheme for clustering dynamic data streams. In *ICMLA '06: Proceedings of the 5th International Conference on Machine Learning and Applications*, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.
- [14] A. J. R. Ruiz and L. M. Ortega. Geometric algorithms on CUDA. In *GRAPP*, pages 107–112, 2008.
- [15] J. Stratton, S. Stone, and W. mei Hwu. Mcuda: An efficient implementation of CUDA kernels for multi-core CPUs. In *21st Annual Workshop on Languages and Compilers for Parallel Computing (LCPC'2008)*, July 2008.
- [16] J. A. Stuart and J. D. Owens. Message passing on data-parallel architectures. *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS*, 2009.
- [17] R. Wu, B. Zhang, and M. Hsu. Clustering billions of data points using GPUs. In *UCHPC-MAW '09: Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop*, pages 1–6, New York, NY, USA, 2009. ACM.
- [18] Y. Zhang, F. Mueller, X. Cui, and T. Potok. GPU-accelerated text mining. In *Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods*, Mar, 2009.