International Journal of Grid and Distributed Computing
Vol. 9, No. 11 (2016), pp.01-16
http://dx.doi.org/10.14257/ijgdc.2016.9.11.01

A Self-adaptive Workload Balancing Algorithm on GPU Clusters

Jianjiang Li[1], Yajun Liu[2], Peng Zhang[3,*], Qingsong Miao[4], Lei Zhang[5] and Wei Chen[6]

[1-6]*Department of Computer Science and Technology, University of Science and Technology Beijing, Beijing 100083, P.R.China*
[1]*lijianjiang@ustb.edu.cn*, [2]*liuyajun1992china@163.com*,
[3]*chinazhangpeng1992@163.com*, [4]*897160410@qq.com*, [5]*melodywhan@126.com*,
[6]*chenweichina1992@163.com*

Abstract

With the wide application of GPU in High Performance Computing, more and more heterogeneous CPU+GPU clusters have been established in many fields. But with the comprehensive using of heterogeneous CPU+GPU clusters, workload balancing has become an important problem when the process nodes coordinate with each other, and the execution time of a program on imbalanced clusters resides on the slowest node. Although there are many strategies and algorithms that can solve the problem of workload balancing to some extent, they generally face the problem of high consumption of communication caused by the task migration. In order to make up for the existing deficiencies, this paper proposes a virtual task migration algorithm adapted to GPU clusters on CUDA platform. This algorithm uses virtual task migration to avoid actual data transmission between nodes, so the communication overhead is obviously decreased. At last, this paper performs an actual test using matrix multiplication to verify this algorithm. The experiment results show that compared with static task partitioning, the algorithm proposed in this paper can effectively achieve dynamic workload balancing and reduce the execution time of programs on GPU clusters, thus the algorithm can significantly improve program execution performance of GPU clusters on CUDA platform.

Keywords: *GPU Clusters; Dynamic Workload Balancing; Task Migration; CUDA*

1. Introduction

With the development of science and society, the need of High Performance Computing increases rapidly, and High Performance Computing has become an important research to break through the limitations of Moore theorem. The powerful parallel computing ability of GPU attracts many researchers.

However, when calculating the multi-dimensional and multi-scale data, the computing pattern of GPU has some limitations. Firstly, the original design purpose of GPU is to accelerate the graphics drawing, so developers need to access GPU from API such as OpenGL or DirectX, which requires developers not only need to learn some graphics programming, but also know how to convert common computing problems into graphics computing problems. Secondly, GPU's computing architecture has big differences from multi-cores CPU, and GPU pays more attention on data parallel computing, namely executing same computation on different data, while it has insufficient on the mutual exclusive, synchronization and atomicity. These factors limit GPU's applied range of general purpose computing.

Peng Zhang is the corresponding author.

ISSN: 2005-4262 IJGDC

In 2007, NVIDIA released CUDA [1]. CUDA architecture has solved above-mentioned problems, and designs a new architecture for GPU computing. Under CUDA architecture, developers can program on GPU using CUDA C. CUDA C is an extension of standard C and it's easy to study and use, what is more, its biggest advantage is that developers don't need to learn the knowledge of graphics.

As NVIDIA releases more new GPU products which support CUDA and more institutions build heterogeneous CPU+GPU clusters, using GPU to make parallel computing is becoming more important. For example, "TianHe 2" uses heterogeneous CPU+GPU architecture and has 32000 CPUs and 48000 GPUs, in addition to, the total number of cores reaches to 3120000.

Due to various historical and realistic reasons, heterogeneous computing is still facing many problems, mainly in scalability, workload balancing, adaptability, communication, memory, ***etc***. Workload balancing has become an important problem when the process nodes coordinate with each other. It is an effective way to enhance the system utilization, but the execution time of a program on imbalanced clusters is constrained by the slowest node.

Workload balancing algorithm is closely associated with task scheduling. According to the scheduling time, task scheduling can be divided into static scheduling and dynamic scheduling. Using static scheduling, tasks will be distributed to processors according to the characters of each task and processor, and after which there will be no re-distribution. Static scheduling corresponds to static workload balancing. Dynamic scheduling achieves dynamic workload balancing by analyzing workload of each processor and migrating tasks from heavy load processors to light load processors. Currently, there are many dynamic workload balancing algorithms on CPU clusters. For example, CPU or memory-centric load balancing schemes, which suffer significant performance drop under I/O-intensive workloads due to the imbalance of I/O load. There are few researches on heterogeneous GPU+CPU clusters, and dynamic workload balancing problems also occur on heterogeneous GPU+CPU clusters.

To achieve workload balancing on heterogeneous GPU+CPU clusters, this paper proposes a virtual task migration algorithm which is suitable for the GPU clusters on CUDA platform. The algorithm is able to avoid actual data transmission between nodes in the process of task migration, therefore, the communication overhead is significantly reduced, which not only optimizes the performance of communication between nodes but also improves the ability of computation on the GPU clusters. The majority of data are multidimensional, and matrix is a kind of typical structure of presenting multidimensional data, which is convenient to calculate, easy to store and easy to design. So this paper chooses a way that using matrix calculation to test the feasibility of the algorithm, which is more closed to the actual environment in which complex numerical calculation can be done using GPU clusters.

The rest of this paper is organized as follows: section 2 introduces related works about workload balancing on GPU clusters.

Section 3 introduces the reasons of workload imbalance, then uses two groups of matrix multiplication to illustrate the phenomenon of workload imbalance and its influence on the system uptime.

Section 4 proposes a virtual task migration algorithm adapted to GPU clusters. This algorithm uses virtual task migration to avoid actual data transmission between nodes and performs dynamic task scheduling between different nodes, so the communication overhead is obviously decreased and self-adaptive workload balancing on GPU clusters is achieved.

Section 5 realizes the virtual task migration algorithm using matrix multiplication and tests the two groups of matrix multiplication which are in section 3, then the contrastive results demonstrate the effectiveness of the virtual task migration algorithm proposed in this paper.

Section 6 is the conclusion of the virtual task migration algorithm.

2. Related Work

How to effectively explore the power of CPU/GPU heterogeneous clusters has been widely researched by many people. The increase of computational power of programmable GPU (graphics processing unit) brings new concepts for using these devices for generic processing. Hence, using the CPU and GPU for data processing causes new ideas of dealing with distribution of tasks among CPU and GPU, such as automatic distribution.

In paper [2], the authors propose GROPHECY, a GPU performance projection frame work that can estimate the performance benefit of GPU acceleration without actual GPU programming or hardware. Users need only to skeletonize pieces of CPU code that are targets for GPU acceleration. Paper [3] proposes a new scheduling and workload balancing scheme, HDSS, for execution of loops having dependent or independent iterations on heterogeneous multiprocessor systems. The new algorithm dynamically learns the computational power of each processor during an adaptive phase and then schedules the remainder of the workload using a weighted selfscheduling scheme during the completion phase. In [4], the authors develop a multi-level partitioning and distribution method that guarantees a near-optimal communication volume. They have also extended heterogeneous tile algorithms to work on distributed memory GPU clusters. The main idea is to execute a serial program and generate hybrid-size tasks, and follow a dataflow programming model to fire the tasks on different compute nodes. Then they devise a distributed dynamic scheduling runtime system to schedule tasks, and transfer data between hybrid CPU-GPU compute nodes transparently. The authors in paper [5] propose a load balancing method based on a semidefinite optimization. They hope that this method, coupled with a multi-layered programming, can perform a HPL benchmark on CPU and GPU clusters and HPC Cloud systems are more efficiently than methods used today. In paper [6], the authors propose a flexible energy efficient task scheduling scheme for heterogeneous tasks in the heterogeneous GPU-enhanced clusters. A system model and a task model for the heterogeneous clusters are formulated in the paper. According to the node selection policy based on GPUs utilization of the particular task, it can decrease the static energy consumption of GPUs in idle status. By the division of task types and buddy allocation, it can improve the utilization of the CPU resource to increase energy efficiency. In [7], they treat a heterogeneous system as a distributed-memory machine, and use a heterogeneous multi-level block cyclic distribution method to allocate data to the host and multiple GPUs to minimize communication. They can achieve a high degree of parallelism, minimized synchronization, minimized communication, and load balancing. However, the approach in this paper, the largest input is constrained by the memory capacity of each GPU. A programmer is responsible for specifying which piece of code should be executed on a GPU. Then its runtime can execute the annotated code in parallel on the host and GPUs. Charm++ is an object-oriented parallel language that uses a dynamic load balancing runtime system to map objects to processors dynamically [8]. In [9], they examine explicit partitioning of data objects and its effects on operation partitioning. The partitioning of data objects must consider several factors: object size, access frequency pattern, and dependence patterns between operations that manipulate the objects. This work proposes a compiler-directed approach to synergistically partition both data objects and computation across multiple clusters. However, their work dose not focus on partitioning data objects for cache systems. In order to handle a cache memory system, the partitioning algorithm must be extended to deal not only with communication patterns between data and computation operations, but also with the data usage patterns over time, as objects can be moved into and out of the caches. In [10], they make two contributions in privacy-preserving data mining. First, they introduce the concept of arbitrarily

partitioned data, which is a generalization of both horizontally and vertically partitioned data. Second, they provide an efficient privacy-preserving protocol for k-means clustering in the setting of arbitrarily partitioned data. However, this algorithm proposed in this paper potentially leaks some information through the intermediate cluster assignments, even though the intermediate cluster centers themselves are not revealed. In [11], a highly decentralized, distributed and scalable algorithm for scheduling jobs and a method of balancing the load across the resources in heterogeneous computational grid environments are presented, which can minimize response time of the jobs that arrive at a grid system for processing. But load balancing strategies cannot be embedded into real world grid computing environments.

3. Workload Imbalance

Currently, when solving a problem on GPU clusters, the first thing to do is to divide problem data according to the computing power of different nodes, then distributes data to different nodes to perform computation, at last collects computed results. However, this situation is only suitable for problems such as encryption, decryption, dense linear algebra and so on. The character of these problems is that when processing units data, the effective amount of computation are same and good workload balancing can be achieved on these problems. However, when making some calculation, according to the different dimensions of the data, the effective amount of computation for units data procession is different in some problems, such as three-dimensional matrix, sparse matrix multiplication and array quick sort. In such cases, if the algorithm divides data only according to the computing power of GPU, the problem of workload imbalance will arise, and the execution time of the whole program is constrained by the GPU which needs the longest computing time. As shown in Figure 1, the execution time of p sub-tasks on GPU clusters are $T_0, T_1,...,T_{p-1}$, then the final time of the entire task T is the maximum value of $\{T_0, T_1,...,T_{p-1}\}$.

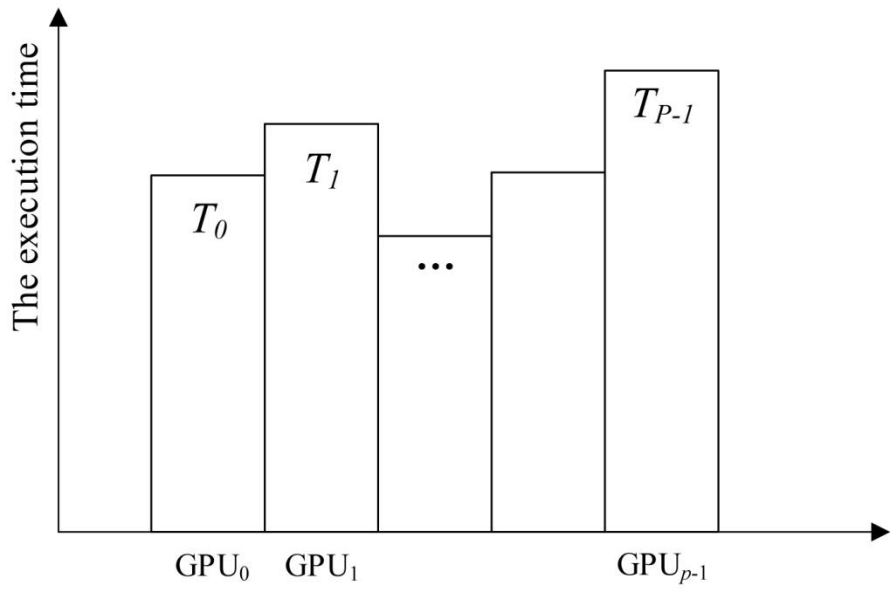


Figure 1. Computing Time is Constrained by the Slowest Node When Workload Is Imbalanced

To a great extent, reliability depends on the computer hardware and software which have been reasonably distributed. The target of workload balancing is just trying to make all nodes with different computing power complete their own task with identical time.

Partition is the process of slicing computation and data, and data partition is especially important for GPU clusters. A dynamic data partition method partitions data according to the computation power of each GPU node. For parallel programs, data partition is more important, and it directly decides the final execution efficiency of programs.

The computation on GPU clusters needs data transfer and communication between different nodes, and the communication mode in this paper is MPI [12]. MPI is good at partitioning the computing tasks and the communication between nodes, meanwhile CUDA is good at executing these parallel parts.

The matrix multiplication on GPU clusters uses two GPUs, and the specific configuration is shown in Table 1. We can see from Table 1 that GPU1's memory storage and core frequency are obviously higher than GPU0's and its computation power is about 3 times of GPU0. There are 4 groups of matrix, where A1×B1 and A3×B3 are regular matrices, and A2×B2 and A4×B4 are sparse matrices and the sparse degree is 50%. In addition, matrix C is the result matrix.

Table 1. Conguration of Different GPUs

GPU Type	GeForce 9300MGS	GeForce GT540M
GPU Bit	64Bit	128Bit
GPU Memory	512MB	1024MB
Core Frequency	450MHz	1344 MHz
Number	GPU0	GPU1

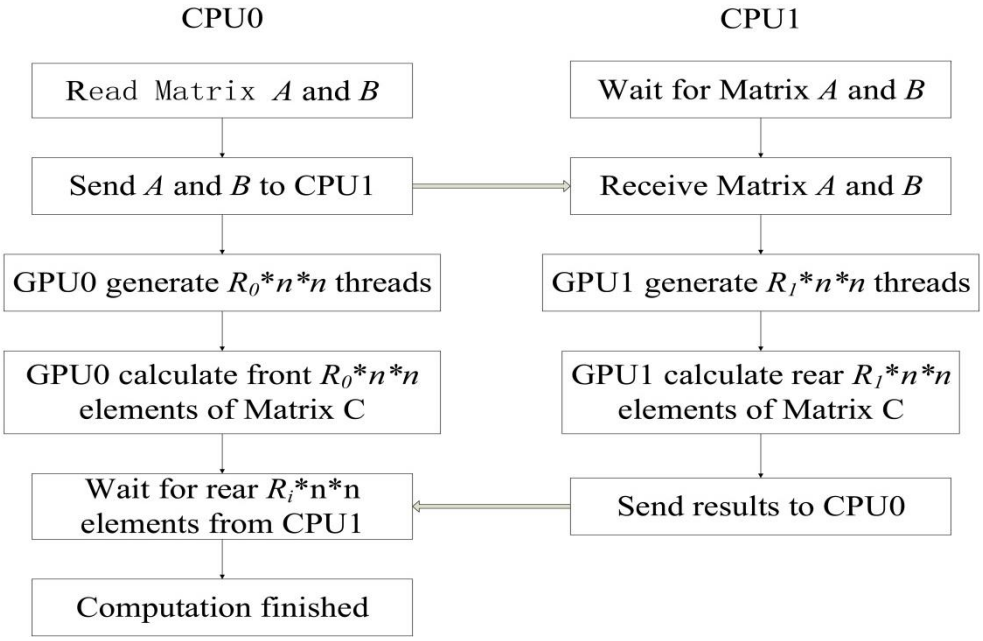


Figure 2. The Flow Diagram of Matrix Multiplication

The ratio of data partition is calculated according to the computation power of two GPUs. This paper uses task scale factor R_i to indicate the ratio of the workload of the *i*th task, so GPU0's scale factor is R0 = 450 / (450 + 1344) = 0.25, GPU1's scale factor is R1 = 1344 / (450 + 1344) = 0.75, therefore the ratio of task partition is 1:3. For matrix multiplication, GPU0 computes the front 25 percent elements of matrix C and GPU1 computes the rear 75 percent elements of matrix C. The flow diagram of matrix multiplication is shown in Figure 2, and the pseudo-code is given in Algorithm 1.

Algorithm 1 Matrix Multiplication on two nodes

Input:
Matrix A and B;
Output:
The result Matrix C;
1: **if** This is node 0 **then**
2: read Matrix A and B from files/standard input;
3: MPI Send(): send matrix A and B to node 1;
4: cudaMalloc(): allocate memory for matrix A and B on GPUs;
5: cudaMemcpy(): copy matrix A and B from CPU to GPU;
6: kernel<<<>>>(): T0*n*n threads to calculate A × B;
7: cudaMemcpy(): copy the computing results from GPU to CPU;
8: cudaFree();
9: wait for node 1 to finish;
10: MPI Recv(): receive the computing results from node 1;
11: **else**
12: //This is node 1
13: MPI Recv(): receive matrix A and B from node 0;
14: cudaMalloc(): allocate memory for matrix A and B on GPUs;
15: cudaMemcpy(): copy matrix A and B from CPU to GPU;
16: kernel<<<>>>(): T1*n*n threads to calculate A×B;
17: cudaMemcpy(): copy the computing results from GPU to CPU;
18: cudaFree();
19: wait for node 0 to finish;
20: MPI Send(): send the computing results to node 0;
21: **return** Matrix C;

The testing results on GPU clusters are shown in Table 2, where the total execution time includes the execution time on GPU nodes, the cost of communication between nodes, and the cost of communication between CPU and GPU within each node. And A2, B2, A4, B4 are sparse matrices.

This paper uses workload scale factor to describe the degree of workload balancing on different GPUs, and workload scale factor is the ratio of GPU time of different nodes. We can see from Table 2 that:

On the GPU clusters with two nodes, the workload balancing scale factor is 315:322 = 0.978 for the small-scale regular matrices A1×B1, and for large-scale regular matrices A3×B3, the workload balancing scale factor is 15802:15688 = 1.007. Both of them are close to 1, which indicates that the degree of workload balancing is very high.

Table 2. Testing Results of Matrix Multiplication on GPU Clusters

Matrix	A1×B1	A2×B2	A3×B3	A4×B4
Matrix Scale	256×256	256×256	1024×1024	1024×1024
Matrix Property	regular	sparse	regular	sparse
The Execution Time on GPU0(ms)	315	143	15802	8458
The Execution Time on GPU1(ms)	322	81	15688	4883
Communication Overhead(ms)	56	53	1104	1028
Total Time(ms)	378	196	16906	9486

For sparse matrices, there is unbalanced workload on the GPU clusters when performing data partition only by computation power: for A2×B2, the workload balancing

scale factor is 143:81 = 1.77; for A4×B4, it is 8458:4883 = 1.73. The different sparse degree of sparse matrices leads to the difference of actual amount of computation. The actual amount of computation on GPU1 is very low, so it finishes its computing task very early, thus the master must wait until the slower GPU0 finishes its own task to get the final results, finally the execution time of the whole program is constrained by the slowest node.

4. Virtual Task Migration Algorithm

The dynamic workload balancing algorithms are based on task migration on multi-processors or clusters. The workload on different processors maybe imbalance, on this occasion, task immigration is needed. Task migration needs certain data communication. The communication overhead on multi-processors is acceptable, but on a cluster or clusters, the frequent task migration could be the bottleneck of the whole system. But the virtual task migration algorithm proposed in this paper does not have actual task migration between nodes, which can effectively save the time of data transmission.

For a GPU cluster with *p* nodes, the flow diagram of virtual task migration algorithm is shown in Figure 3.

Now we will give the detailed steps of the algorithm. The algorithm will be divided into 3 parts: data transmission, task partition, task execution and migration.

Data transmission includes step1, step2 and step9 in Figure 3, and the main job of data transmission is to distribute data for computation to the host memory of all nodes, then each host memory copies data to their own GPU memory, at last each host gathers computed data to its host memory of the master node. These three steps will be introduced in detail as following.

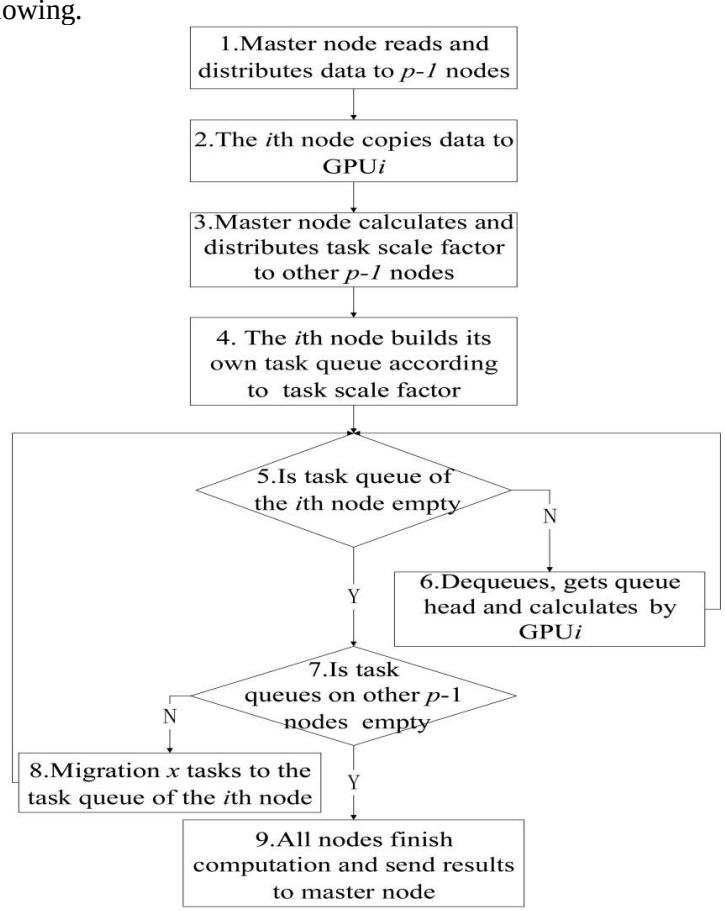


Figure 3. Virtual Task Migration Algorithm on GPU Clusters

Step 1: Reads and distributes data to *p*-1 nodes. The main work of this step is that the master node CPU0 reads or generates and distributes data that will be processed to other *p*-1 nodes. The distribution uses the broadcast operation MPI_Bcast in MPI.

Step 2: The *i*th node copies data to GPU*i*. According to specific problem, function cudaMalloc( ) or cudaMallocPitch( ) could be used to allocate GPU memory to store data that will be processed and computed results. Data copy operation could use function cudaMemcpy() or cudaMemcpy2D(), and the argument "cudaMemcpyKind" in the two memory copy functions should be assigned value "cudaMemcpyHostToDevice" which means that the direction of data copy is from host memory to device memory.

Step 9: All nodes finish computation tasks and copy results to the master node. When task queues on all nodes are empty, the all computation tasks of the whole system are finished, then the computed result of each nodes should be gathered to the master node. The gather operation uses different collective communication functions based on the characters of specific tasks. After calling the gather operation, the master node may need addition data processing to finish the final computation.

Task partition includes step 3 and step 4 in Figure 3, and the main purpose of task partition is to divide the entire task into different sub-tasks based on the computation power of each node and delivers them to each node. The two steps will be introduced in detail as following.

Step 3: Calculates and distributes the task scale factor to other *p*-1 nodes.

The main job of this step is to generate an array R with *p* size according to the computation power of each node. In R, R_i indicates the task scale factor of the *i*th node, and its specific calculation method is as follows:

We assume the computation powers of *p* nodes are c_0, c_1, c_2,..., c_{p-1}, and the whole computation power of the cluster is equation (1) , and $R_i = c_i / CP$. After task scale factor array R is calculated, it will be broadcasted to other *p*-1 nodes through MPI_Bcast operation.

$$CP = \sum_{i=0}^{p-1} c_i \tag{1}$$

Step 4: The *i*th node builds task queue according to task scale factors.

A task can be represented by a struct which contains the data to be processed and data processing methods. In order to implement the virtual task migration algorithm proposed in this paper, the data to be processed is defined as pointers to data, and the data can be accessed by the pointers. The definition of a task is as follows:

```
typedef struct {
  void *data;
  global void *kernel(void *data);
  dim3 gridDim;
  dim3 blockDim;
 }Task;
```

In this struct, "data" stands for the data to be processed or calculated, "kernel" stands for the kernel function used to process data, "gridDim" and "blockDim" stand for the grid dimension and thread block dimension when launching the kernel function. The granularity of task can affect the execution of programs: if the granularity is too coarse, workload imbalance will arise; if the granularity is too fine, the calculation units of GPU can not be fully utilized which will lead to computation inefficiency. TOTAL_TASK_NUM is used to stand for the number of total tasks, and the number of tasks in task queue of nodes *i* are the multiplication of task scale factor R_i and TOTAL_TASK_NUM.

Task queue uses linked queue and its definition is as follows.

```
typedef struct Node {
    Task data; // data element
```

```
  struct Node *next;
}Task;
typedef struct {
  QNode * front; // head of queue
  QNode * rear; // rear of queue
}LinkQueue;
```

Firstly, a task queue will be initialized, namely an empty task_queue will be constructed and its construction method is as follows.

bool InitQueue(LinkQueue * task queue);

After task_queue is constructed, the TOTAL_TASK_NUM * R_i tasks will be queued. The enqueue function is as follows:

bool EnQueue(LinkQueue * task queue, Task task);

The variable "current_task_num" stands for the number of current tasks in task_queue, and its initial value is 0. Once a task is enqueue, the value of current_task_num adds 1. On the other hand, once a task is dequeue, the value of current_task_num minuses 1. When all tasks are queued, step 3 is finished. Before the task is done by GPU, it must make sure that every node has established its own task queue through MPI_Barrier operation.

Task execution and migration include steps from step 5 to 8.The purpose of task migration is to adjust the workload of nodes, make the execution time same of each node possible and improve the integral efficiency.

Step 5: Judge whether the task enqueue of ith node is empty.

There are two ways to judge whether the task queue is empty: firstly, if the value of "current_task_num" is equal to 0, the queue is empty. Secondly, if the head pointer is same as the tail pointer, the queue is empty. Its function prototype is as follows:

bool QueueEmpty(LinkQueue task queue);

If the current task queue is empty, then it should check whether there are unfinished task in other task queues (Step 7). If all tasks have finished, flag "all_finished_flag" will be set as true. If "all_finished_flag" is true, tasks in other nodes are finished (Step 9). Else, current task queue will wait for task migration and get new tasks.

Step 6: This step can be divided into 3 sub-steps: firstly, the head task of the task queue will be obtained. Secondly, dequeue operations will be executed and the value of "current_task_num" will be modified. At last, GPU*i* proceeds computation according to the request of the head task and returns the computing results to the master node. The function prototype getting the head of the task queue is as follows:

bool GetHead(LinkQueue task queue, Task *curTask);

"curTask" stands for the task to be executed, and a dequeue operation is needed after getting this task:

bool DeQueue(LinkQueue *task queue);

After dequeue, GPU*i* will execute computation operation of current task "curTask". The data needed by computation is copied to GPU in Step 4, so there is no need to allocate memory and copy data.

curTask.kerneln<<<curTask.gridDim, curTask.blockDim>>>(curTask.data);

After computing, the results will be copied to the memory of the host node.

Step 7: Check whether task queues on other *p*-1 nodes are empty. When a node finishes its own computation task, it needs to check if there are unfinished tasks in the task queues of other nodes. Firstly, check if "current_task_num" is equal to 0. If it is not true, there are tasks running in the task queue and no need for task migration. If it is true, current node has finished its task, and it needs to collect the size of task queues in other nodes and acquires part of unfinished tasks for its own computation. By reading "current_task_num" in other nodes in sequence, current node can know if they are empty or not.

Step 8: Migrate *x* tasks to the task queue of *i*th node.

If node *i* has finished its computation task, it will check other nodes and find the first node which has unfinished tasks. Taking node *j* for example, *x* tasks will be migrated from the *j*th node to the *i*th node. The value of *x* is calculated according to the computation power of the *i*th node and the *j*th node. The computation of *x* is shown in equation (2).

$$x = current\_task\_num * R_i / (R_i + R_j) \quad (2)$$

Because sending multiple messages wastes more time than sending a message with same amount of data, so when *x* is calculated, it does not carry out *x* times of task migration. By contrast, it will pack and send *x* tasks (It is shown in Figure 4). Firstly, *x* tasks will be obtained from the task queue of the *j*th node and put in buffer "task_transfer_buffer". Secondly, after receiving tasks from the *j*th node, the *i*th node will put these tasks in its own buffer "task_transfer_buffer". Thirdly, the *i*th node executes *x* times enqueue operations and puts *x* tasks in its task queue. Because *x* tasks only contain the address and process function of data, so there is no actual data transmission between nodes, and there is only little communication overhead, that is also why this algorithm is called the virtual task migration algorithm.

There is need to set a threshold value for the number of current tasks. If the value of *x* is less than the threshold value, there is no need to execute task migration. Because there is too less tasks to immigrate, the communication overhead may offset the performance improvement. The following two algorithms give the pseudo-code on the 0th node and other nodes. The algorithm 2 gives the pseudo-code on the 0th node.

Algorithm 2 Tasks on node 0

```
1: read and distribute data to p nodes;
2: calculate and distribute task scale factor to other p-1 nodes;
3: build task_queue according to scale factor;
4: while task_queue is not equal to 0 do
5:     if current_task_num on all nodes is 0 then
6:         all tasks have finished;
7:         break;
8:     else if current_task_num is 0 then
9:         migrate x tasks to the task_queue;
10:    else
11:        dequeue and get queue head from task queue;
12:        complete current task with GPU 0;
13: wait for other p-1 nodes to finish;
14: collect results from other p-1 nodes.
15: return full results;
```

The algorithm 3 gives the pseudo-code on node *i*.

Algorithm 3 Tasks on node *i*

```
1: receive data from node 0;
2: receive scale factor from node 0;
3: build task_queue according to scale factor;
4: while task_queue is not equal to 0 do
5:     if current_task_num on all nodes is 0 then
6:         all tasks have finished;
7:         break;
8:     else if current_ task_num is 0 then
9:         migrate x tasks to current task_queue;
10:    else
11: dequeue and get queue head from task_queue;
12: complete current task with GPU i;
13: wait for node 0 to finish;
```

14: send results to node 0.
15: **return** null;

5. Experimental Results and Analyzation

We has definite applicability and universality to use common matrix operations to test the performance of the virtual task migration algorithm proposed in this paper. The matrix multiplication test platform consists of two nodes: the 0th node and the 1th node. The 0th node configured computing ability is weaker in the NVIDIA GeForce 9000 MGS graphics, core frequency 450MHz, and the 1th node is configured with a strong computing power of the NVIDIA GeForce GT 540M, core frequency 1344MHz.

The specific hardware configuration has been shown in Table 3, and the software environment has been shown in Table 4. For comparison, the matrices used here are matrices in the second chapter: A2, B2 and A4, B4. The two couples of sparse matrices show workload imbalance when the static data partition method is applied. The results are shown in Table 5, Figure 4 and Figure 5.

Table 3. Test Hardware Platform

Hardware Platform			
Node0	Host	CPU0	Intel Pentium T3400
		memory	1G
		hard disk	500G
	Device	GPU0	GeForce 9300MGS
		video memory	512MB
		core frequency	450MHz
Node1	Host	CPU1	Intel Corei5 2430M
		memory	2G
		hard disk	500G
	Device	GPU1	GeForce GT 540M
		video memory	1024MB
		core frequency	1344MHz

Table 4. Test Software Platform

Software Environment	
NVIDIA toolkit	CUDA toolkit_3.1_linux
NVIDIA SDK	CUDA sdk_3.1_linux
Complier	GUN GCC 4.8.2
MPI version	MPICH 2.0
Operating system	Ubuntu 13.04

Table5. Testing Results of Matrix Multiplication on GPU Clusters

Matrix	A2×B2	A2×B2	A4×B4	A4×B4
Algorithm	Static Partition	Virtual Task Migration	Static Partition	Virtual Task Migration
Matrix Scale	256×256	256×256	1024×1024	1024×1024
The Execution Time on GPU0(ms)	143	118	8458	6770
The Execution Time on GPU1(ms)	81	106	4883	5595

Workload Balancing Ratio	1.77	1.11	1.73	1.21
Communication Overhead(ms)	53	74	1028	1775
Total Time(ms)	196	192	9486	8545

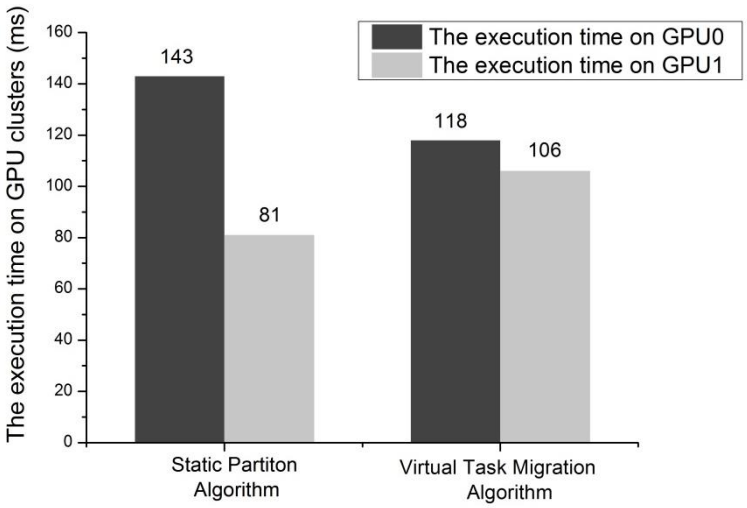


Figure 4. The Execution Time of 512×512 Matrix Multiplication on GPU Clusters

Table 5, Figure 4 and Figure 5 show that, for small scale matrix multiplication A2×B2, the virtual task migration algorithm proposed in this paper has reduced the workload balancing ratio from original 1.77 to 1.11 and obtains well workload balancing, but the final acceleration effect is not good because of the increasing communication overhead. For large scale matrix multiplication A4×B4, the virtual task migration algorithm proposed in this paper has reduced the workload balancing ratio from original 1.73 to 1.21 and also obtains well workload balancing. Although the communication overheads also increases, but the total execution time of this algorithm decreases by 10 percent, which demonstrates the effective of the virtual task migration algorithm proposed in this paper.

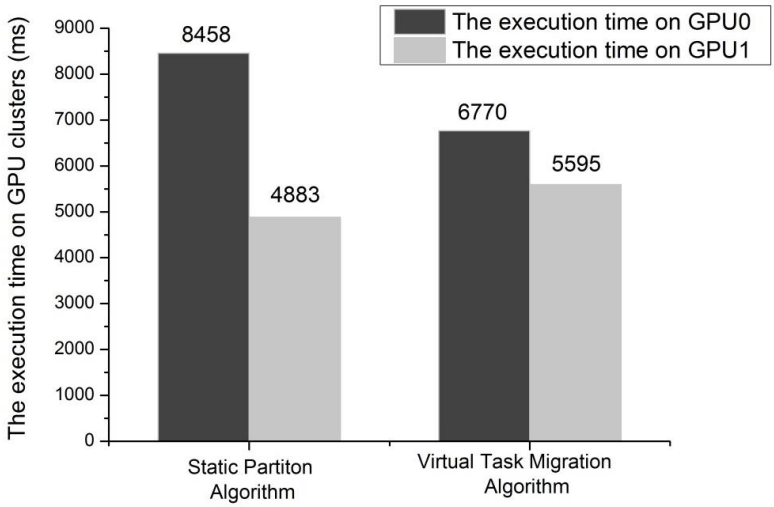


Figure 5. The Execution Time of 1024×1024 Matrix Multiplication on GPU Clusters

Because the virtual task migration algorithm is implemented in GPU clusters, the communication overhead is inevitable. When data scale is small, the computation cost is also small, and the communication cost is relatively low compared with computation overhead, so

The performance improvement is not obvious. As the data scale growths, the improvement of GPU utilization has exceeded the increase of communication overhead, especially for programs such as matrix multiplication whose computation cost has super linear growth with the growth of data scale.

6. Conclusion

In order to reduce the communication overhead of the migration in the GPU clusters and effectively improve the performance of load balancing, this paper proposes a virtual task migration algorithm adapted to GPU clusters on CUDA platform, solves the workload balancing problem among nodes in GPU clusters and performs experiments with matrix multiplication to verify its effectiveness at last.

The biggest trait of the virtual task migration algorithm proposed in this paper is that it can achieve virtual task migration between different nodes within little communication overhead and realize dynamic workload balancing. For large-scale data operation calculation, the virtual task migration algorithm proposed in this paper compared with the traditional static load has an obvious improvement in performance: reduce the workload balancing ratio from original 1.73 to 1.21 and also obtain well workload balancing, and the total execution time of this algorithm decreases by 10 percent, which demonstrates the effective of the virtual task migration algorithm proposed in this paper. In this article, however, we did not consider the operation time of calculation in detail when the computation is not big enough. It also needs to study of the granularity of task partitioning, and we need to be further solve all these problems.

Acknowledgment

This work was supported in part by the National High Technology Research and Development Program of China (No.2015AA01A303) and Beijing Key Subject Development Project (XK10080537).

References

[1] The official nvidia website, http://www.nvidia.cn/page/home.html, **(2013)**.
[2] J. Meng, V. A. Morozov, K. Kumaran, V. Vishwanath and T. D. Uram, “Grophecy: Gpu performance projection from cpu code skeletons”, Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, **(2011)**, p. 14.
[3] M. E. Belviranli, L. N. Bhuyan and R. Gupta, “A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures”, ACM Transactions on Architecture and Code Optimization (TACO), vol. 9, no. 4, **(2013)**, p. 57.
[4] F. Song and J. Dongarra, “A scalable framework for heterogeneous gpu-based clusters”, Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures, ACM, 2012, pp. 91-100.
[5] D. Tomić and D. Ogrizović, “Running high performance linpack on cpugpu clusters”, MIPRO, 2012 Proceedings of the 35th International Convention, IEEE, **(2012)**, pp. 400-404.
[6] H. Huo, C. Sheng, X. Hu and B. Wu, “An energy effcient task scheduling scheme for heterogeneous gpu-enhanced clusters”, Systems and Informatics (ICSAI), 2012 International Conference on, IEEE, **(2012)**, pp. 623-627.
[7] F. Song, S. Tomov and J. Dongarra, “Enabling and scaling matrix computations on heterogeneous multi-core and multi-gpu systems”, Proceedings of the 26th ACM international conference on Supercomputing, ACM, **(2012)**, pp. 365-376.
[8] P. Jetley, L. Wesolowski, F. Gioachin, L. V. Kalé and T. R. Quinn, “Scaling hierarchical n-body simulations on gpu clusters”, Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE Computer Society, **(2010)**, pp. 1-11.

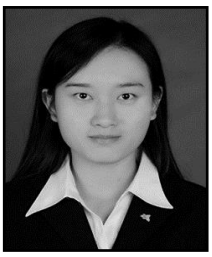
[9] M. L. Chu and S. A. Mahlke, "Compiler-directed data partitioning for multicluster processors", Proceedings of the International Symposium on Code Generation and Optimization, IEEE Computer Society, **(2006)**, pp. 208-220.
[10] G. Jagannathan and R. N. Wright, "Privacy-preserving distributed k-means clustering over arbitrarilypartitioned data", Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining, ACM, **(2005)**, pp. 593-599.
[11] M. Nandagopal, K. Gokulnath and V. R. Uthariaraj, "Sender initiated decentralized dynamic load balancing for multi cluster computational grid environment", Proceedings of the 1st Amrita ACM-W Celebration on Women in Computing in India, ACM, **(2010)**, p. 63.
[12] The official mpi forum website, http://www.mpi-forum.org/, **(2013)**.

Authors

Jianjiang Li, he is currently an associate professor at University of Science and Technology Beijing, China. He received his PhD degree in computer science from Tsinghua University in 2005. He was a visiting scholar at Temple University from Jan. 2014 to Jan. 2015. His current research interests include parallel computing, cloud computing and parallel compilation.

Yajun Liu, he is currently an master degree candidate at University of Science and Technology Beijing, China. She received her B.S. Degree in computer science and technology from Shanxi University in 2015. Her current research interests include parallel computing, cloud computing and parallel compilation.

Peng Zhang, he is currently a master degree candidate in University of Science and Technology Beijing, China. He received his B.S.Degree in information management and information system from Tangshan Normal University in 2015. His current research interests include parallel computing, cloud computing and parallel compilation.



Wei Chen, she is currently an master degree candidatate University of Science and Technology Beijing, China. She received her B.S.Degree in Network engineering from North China Institute of Aerospace Engineering in 2014. Her current research interests include parallel computing, cloud computing and parallel compilation.

