All-pairs Shortest Path Algorithm based on MPI+CUDA Distributed Parallel Programming Model

Wu Qingshuang
College of Territorial Resources and Tourism, Anhui Normal University, Wuhu, Anhui, 241003, China
Email: qepwq_wu@yeah.net

Tong Chunya
College of Electronic and Information Engineering, Ningbo University of Technology, Ningbo, Zhjiang, 315016, China

Wang Qiang
City Construction Academy, Hunan City University, Yiyang, Hunan, 413000, China

Cheng Xiangfu
Anhui Key Laboratory of Natural Disaster Process and Prevention, Wuhu, Anhui, 241003, China

***Abstract*—In view of the problem that computing shortest paths in a graph is a complex and time-consuming process, and the traditional algorithm that rely on the CPU as computing unit solely can't meet the demand of real-time processing, in this paper, we present an all-pairs shortest paths algorithm using MPI+CUDA hybrid programming model, which can take use of the overwhelming computing power of the GPU cluster to speed up the processing. This proposed algorithm can combine the advantages of MPI and CUDA programming model, and can realize two-level parallel computing. In the cluster-level, we take use of the MPI programming model to achieve a coarse-grained parallel computing between the computational nodes of the GPU cluster. In the node-level, we take use of the CUDA programming model to achieve a GPU-accelerated fine grit parallel computing in each computational node internal. The experimental results show that the MPI+CUDA-based parallel algorithm can take full advantage of the powerful computing capability of the GPU cluster, and can achieve about hundreds of time speedup; The whole algorithm has good computing performance, reliability and scalability, and it is able to meet the demand of real-time processing of massive spatial shortest path analysis.**

***Index Terms*—Shortest Path Problem; GPU Cluster; Floyd-Warshall Algorithm; Parallel Computing; MPI; CUDA**

I. INTRODUCTION

The shortest path problem is one of most fundamental problem in network analysis, and can affect a variety of meaningful problems: network routing, connectivity analysis, resource allocation, flow analysis, and so on. From the point of view of the graph, the so-called shortest path problem is to find a minimum obstruct intensity path between two nodes in the specified network. According to the different definition of obstruct intensity, the shortest path can not only be refers to the shortest distance in the general geographic sense, but also can be extended to other meaning, such as lowest cost, shortest time, fastest path, and so on.

The shortest path problem is not only a research hot spots in geographic information science, but also is an issue of concern in computer science, operations research, distributed computing, bioinformatics and so on. Many scholars have already conducted an in-depth study on the shortest path problem and proposed a variety of different algorithms. All these existing algorithms achieve good results and have different characteristics, but almost of them solely use the CPU as computing unit, and the time efficiency of the algorithms almost reach the theoretical limit.

The most well-known algorithm solving shortest path for the case of graphs with nonnegative edges is Dijkstra algorithm, which was given by Dijkstra in 1959 [1]. Dijkstra algorithm is inherently sequential since its efficiency depends on a fixed ordering of the vertices, and nearly all the subsequent proposals are based on it. In spite of its early formulation, this classic solution is still presented in almost every textbook on algorithms.

The Bellman-Ford algorithm allows all vertices to be considered in parallel, but at the cost of being not efficient [2]. Provides the details of different formulations of parallel algorithms for the shortest path problem [3]. present a specific proposal for incorporating parallelism into Dijkstra algorithm by using parallel priority queues. However, the literature contains few experimental studies on parallel algorithms of the nonnegative shortest path problem. Some of the more recent works study the use of supercomputers for solving shortest path problem in large graphs [4]. reports performance results on the multithread parallel computer Cray MTA-2, using the Δ-stepping parallel algorithm, it exhibits remarkable parallel speedup

doi:10.4304/jnw.8.12.2797-2803

when compared to competitive sequential algorithms, for low-diameter sparse graphs of 100 million vertices and 1 billion edges.

However, With the problem of solving the shortest path constantly expanding, the amount of data needed to be deal with has a steady increase. Some modern applications, such as network routing, resource allocation, etc, require large graphs with millions of vertices, and some of the previous algorithms become impractical, when we do not have a very expensive hardware at our disposal. Fortunately, GPU cluster supply a high parallel computation power at a low price.

Modern GPU provide tremendous memory bandwidth and computational horsepower. Each new generation of GPUs provides flexible programmability and computational power which exceeds previous generations. Recent high-performance computing-optimized GPUs contain up to 4GB of on-board memory, and are capable of sustaining memory bandwidths exceeding 100GB/sec. The massively parallel hardware architecture and high performance of floating point arithmetic and memory operations on GPUs make them particularly well-suited to many of the same scientific and engineering workloads that occupy GPU clusters, and it has rapidly evolved to become high performance accelerators for data parallel computing [5-7]. The GPU cluster's rapid increase in both programmability and capability has spawned a research community that has successfully mapped a broad range of computationally demanding, complex problems to the GPU cluster [8-10].

To take use of the overwhelming computing power of the GPU cluster to speed up the shortest path problem computing, in this paper, we present an all-pairs shortest paths algorithm using MPI+CUDA hybrid computing model. This proposed algorithm can combine the advantages of MPI and CUDA programming model, and has good computing performance, reliability and scalability. The experimental results show that the MPI+CUDA-based parallel algorithm can take full advantage of the powerful computing capability of the GPU cluster, and can achieve about hundreds of time speedup; it can meet the demand of real-time processing of massive spatial shortest path analysis.

The remainder paper is organized as follows: in section 3, we discuss the basic idea of the distributed parallel programming model based on MPI+CUDA. In Section 3, we introduce the principle of the Floyd-Warshall shortest path algorithm and analyze the feasibility of mapping the shortest path algorithm program onto the GPU cluster by using the MPI+CUDA hybrid programming model, and then we provide the details of implementing our parallel bilinear shortest path algorithm using MPI+CUDA programming model. In Section 4, we demonstrate the strength of our GPU cluster implementation method by measuring the performance over the traditional CPU implementation.

II. PROGRAMMING MODEL

A. MPI

MPI (Message Passing Interface) is a specification for a standard library for message passing that was defined by the MPI Forum, a broadly based group of parallel computer vendors, library writers, and applications specialists [11]. MPI addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process. MPI is a specification, not an implementation; there are multiple implementations of MPI. MPI is not a language, and all MPI operations are expressed as functions, subroutines, or methods, according to the appropriate language bindings, which for C, C++, Fortran-77, and Fortran-95, are part of the MPI standard.

The advantage for the user is that MPI is standardized on many levels. For example, since the syntax is standardized, you can be sure your MPI code will execute under any MPI implementation running on your architecture. Since the functional behavior of MPI calls is also standardized, your MPI calls should behave the same regardless of the implementation, thus guaranteeing the portability of your parallel programs. Performance, however, may vary from implementation to implementation.

In a distributed memory communication environment in which the higher level routines and/or abstractions are built upon lower level message-passing routines the benefits of standardization are particularly apparent. Furthermore, the definition of a message-passing standard, such as that proposed here, provides vendors with a clearly definition base set of routines that they can implement efficiently, or in some cases provide hardware support for, thereby enhancing scalability. As a most widely used parallel computing platform, MPI has good portability and ease of use. It provides a clear definition of the parallel computing program libraries, so that the manufacturers can effectively achieve these program libraries, or in some cases provide hardware support for the program library [12].

The goal of the Message-Passing Interface simply stated is to develop a widely used standard for writing message-passing programs. As such the interface should establish a practical, portable, efficient, and flexible standard for message passing. The framework structure of MPI program is shown in figure 1.

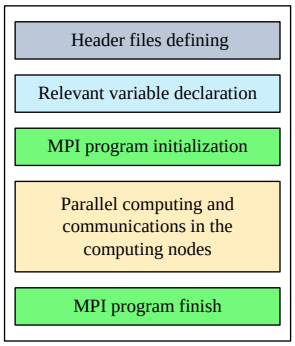

Figure 1. The framework structure of MPI program.

B. CUDA

The multi-core CPU and the many-core GPU are the two most important processors in the current computer. In

tradition, the GPU is only used to process the image rendering tasks, and all the other tasks are processed by the CPU. As a general-purpose processor, the CPU is designed to meet the needs of all different type tasks, most of its on-chip transistors are complex control logic units and large-capacity cache, the actual computing unit is very little [13]. To improve the computing performance, there are two ways: improving the CPU clock speed or increasing the number of the processing cores. Here, it is necessary to add the number of transistors on the CPU chip, and it becomes more and more hard.

GPU originally designed for computer video cards has emerged as the most powerful chip in a high-performance workstation. Unlike multicore CPU architectures, which currently ship with 2-8 cores, GPU architectures are "many-core" with hundreds of cores capable of running thousands of threads in parallel. Over the past few years, a high-end GPU's floating point processing performance can reach 10 times than the same period high-end GPU. And memory bandwidth can reach about five times than the same period desktop platform. To provide the same computing ability, the cost and power consumption of the GPU-based system should be far less than the CPU-based system. Most of the GPU's on-chip transistors are arithmetic logic units, the number of control logic and cache units is very small, so it is suitable for a specifically class of applications with the following characteristics.

1) The degree of computational complexity is very high. The GPUs are designed for the real-time rendering of billions of pixels which each pixel requires hundreds or more operations. It can deliver an enormous amount of computing performance to satisfy the demand for complex real-time applications.

2) The applications are very suitable for parallel processing. The GPU's graphics pipeline is well suited for parallelism. Operations on vertices and fragments are well matched to fine-grained closely coupled programmable parallel compute units, which in turn are applicable to many other computational domains.

3) Throughput is emphasized. GPU implementations of graphics pipeline prioritize throughput over latency. The graphics pipeline is quite deep, perhaps hundreds of thousands of cycles with thousands of primitives in flight at any given time. The pipeline is also feed-forward, removing the penalty of control hazards, further allow in optimal throughput of primitives through the pipeline. This emphasis on throughput is characteristic of applications in other areas as well [7].

According to the characteristics of the GPU's hardware architecture, it can be concluded that the most appropriate application of GPU is the data-intensive and computing-intensive tasks which use a large number of threads on a lot of data in the same calculation processing. Because these tasks' logic branch is very simple, they can take full advantage of the GPUS's processing core. And the delay of reading and writing the memory data is also overshadowed by the efficient intensive computing. With the increasing programmability of the GPU, the applications of using the GPU as general-purpose computing become more and more.

At present, NVIDIA and ATI which are the two main GPU manufacturers have launched their own GPGPU solution each other. These solutions can run the users' code which is designed for the problem domain on the GPU directly. In contrast, NVIDIA's CUDA (Compute Unified Device Architecture) solution is more mature. The NVIDIA CUDA technology is the new software architecture that exploits the parallel computational power of the GPU. When executing CUDA programs, the GPU operates as coprocessor to the main CPU. The GPU handles the core processing on large quantities of parallel information while the CPU organizes, interprets, and communicates information. Compute-intensive portions of applications that are executed many times, but on different data, are extracted from the main application and compiled to execute in parallel on the GPU.

As it is showed in figure 2, CUDA includes three major components: new features on the GPU to efficiently execute programs with parallel data; a C compiler to access the parallel computing resources on the GPU; and a runtime driver dedicated to computing. The ability to interpret and manipulate massive amounts of information is the frontier of computing and with the widely available CUDA SDK any application can exploit the power of the GPU. With the combination of CUDA software and GPUs, developers now have the ability to bring the power of large-scale supercomputer to the desktop and dramatically increase the capability of server clusters [14].

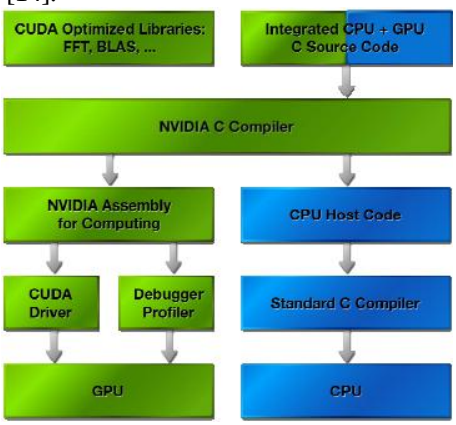

Figure 2. CUDA Software Stack.

The key to CUDA is the C compiler for the GPU. This first-of-its-kind programming environment simplifies coding parallel applications. Using C, a language familiar to most developers, allows programmers to focus on creating a parallel program instead of dealing with the complexities of graphics APIs. As showed in Figure 3, a CUDA program can be divided into two parts: the host-side code and the device-side code. Typically, the host-side code executes on the CPU serially, and device-side code, called kernel, runs on the GPU [15-16]. A kernel executes a scalar sequential program across a set of parallel threads. The programmer organizes these threads into thread blocks; a kernel thus consists of a grid of one or more blocks. A thread block is a group of concurrent

threads that can cooperate amongst themselves through barrier synchronization and a per block shared memory space private to that block. Each thread is block shared memory space private to that block. Each thread is given a unique integer index within its block–via the threadIdx identifier and each block is given a unique integer index–via blockIdx. When launching a kernel, the programmer specifies both the number of blocks and the number of threads per block to be created when launching the kernel. These dimensions are available to the kernel via the identifier's gridDim and blockDim, respectively.

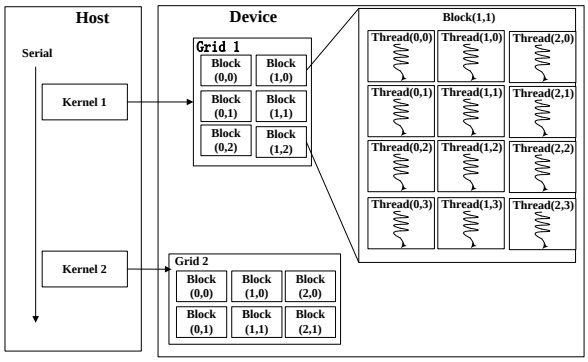

Figure 3. The CUDA programming model.

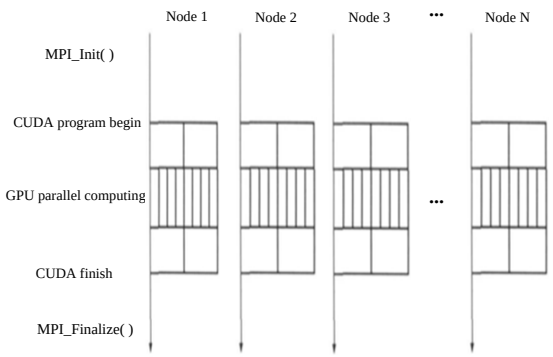

Figure 4. The MPI + CUDA-based distributed parallel computing program model.

C. MPI+CUDA-based Distributed Parallel Programming Model

Many of the GPU cluster applications have been implemented using MPI and CUDA distributed parallel programming model for parallelizing the application. The simplest way to start building an MPI application that uses GPU-accelerated kernels is to use NVIDIA's nvcc compiler for compiling everything.

The basic idea of the distributed parallel programming model is to use MPI+CUDA to realize two-level parallel computing. In the cluster-level, we take use of the MPI programming model to achieve a coarse-grained parallel computing between the computational nodes of the cluster. In the node-level, we take use of the CUDA programming model to achieve a GPU-accelerated fine grit parallel computing in each computational node internal. As shown in figure 4, the process of parallel computing based on MPI + CUDA can be described as below: (1) Input the data set on the master node, then initialize the computing task and the list of subtasks; (2)MPI parallel computing initialization. The master node allocate the subtasks which can be processed in parallel to the relevant computational nodes; (3) In each computational nodes, use CUDA parallel model to deal with the allocated subtask; (4) Each computational nodes return the result of the relevant subtask; (5) The master node get all the results of each computational nodes, then finish the MPI parallel processing and aggregate to generate the final total results [17-19].

III. PROPOSED SCHEME

A. Floyd-Warshall Shortest Path Algorithm

The Floyd-Warshall algorithm is an all-pair shortest paths algorithm, let $G=(V, E)$ be a directed graph with $|V|$ vertices, then we can solve all-pair shortest path by the Floyd-Warshall algorithm using a dynamic programming approach on a directed graph [20].

We store the nodes and edges of the graph in a matrix form. The vertices of the graph are represented by the unique indices along n×n matrix (n represents the number of nodes. The unique indices along n×n matrix can be given by:

$$arc[i][j]=\begin{cases} w_{ij}, if\ (i\neq j) and ((v_i,v_j) or <v_i,v_j>\in E) \\ 0, if\ (i=j) \\ \infty, other \end{cases} \quad (1)$$

where w_{ij} is the weight of edge (v_i,v_j) or edge $<v_i,v_j>$.

Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j, which intermediate nodes are $\{v_i,...,v_j\}$. If $k=0$, the value of $d_{ij}^{(k)}$ is w_{ij}, then the recursive definition of $d_{ij}^{(k)}$ can be given by:

$$d_{ij}^{(k)}=\begin{cases} w_{ij}, & if\ k=0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)}+d_{kj}^{(k-1)}), & if\ k\geq 1 \end{cases} \quad (2)$$

Then pseudocode of the Floyd-Warshall algorithm can be described in figure 5.

```
1: n ← rows [W]
2: D^0 ← W
3: for k ← 1 to n
4:   do for i ← 1 to n
5:     do for j ← 1 to n
6:       if (w[i][k]+w[k][j] < w[i][j]
           then w[i][j] = w[i][k]+w[k][j]
7: return
```

Figure 5. Pseudocode of the Floyd-Warshall algorithm.

In the implementation of the Floyd-Warshall algorithm, it needs to perform three loop, and each loop has n-step, at a time, then time complexity of algorithms is $\theta(n^3)$. Figure 6 shows an example of using the Floyd-Warshall algorithm to solve the all-pair shortest path.

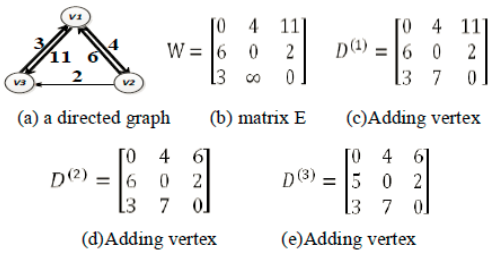

Figure 6. An example of using the Floyd-Warshall algorithm to solve the all-pair shortest path.

B. Implementation of the Parallelism Floyd-Warshall Algorithm Using MPI+CUDA-based Distributed Parallel Programming Model

To begin the algorithm, we partition the matrix into sub-blocks of equal size As it is shown in figure 7, in each computation node, a primary block is set. The primary block for each computation node is along the diagonal of the matrix, starting with the block holding the matrix value (0, 0).

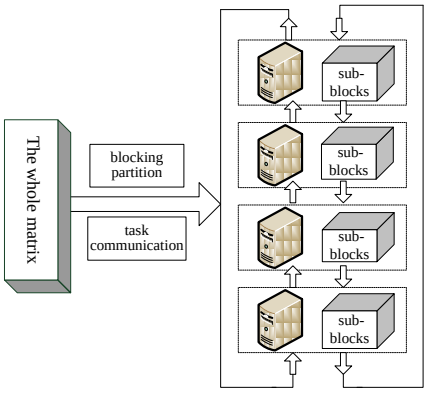

Figure 7. The tasks partition of the parallel algorithm.

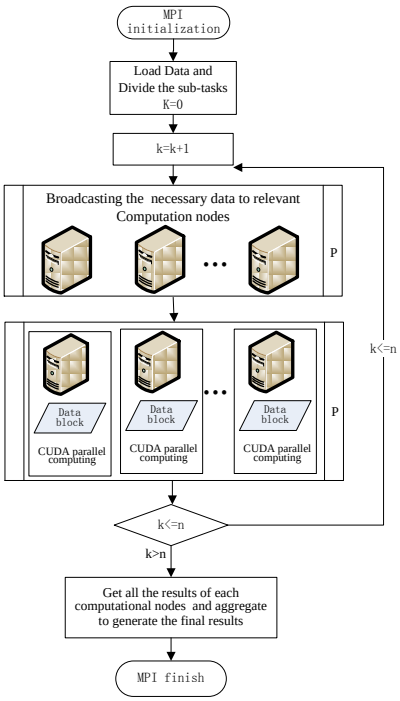

Figure 8. The flow chart of Floyd-Warshall algorithm based on MPI+CUDA.

The process of parallel algorithm based on MPI + CUDA can be described as figure 8: (1)All the computation nodes call MPI_Init ( ) function to initialize; (2)Load the matrix on the master node and partition the matrix into sub-blocks of equal size according the number of the computation node;(3)The master node call MPI_Bcast( ) function to send the sub-blocks to the relevant computational nodes; (4)Perform the k-th iteration cycle; (5)In each computational nodes, utilize the CUDA program to deal with the allocated sub-block; (6)Run the iteration cycle. from (3) to (5) until end of the cycle, then go to the next step; (7) Get all the results of each computational nodes on the master node, then finish the MPI parallel processing and aggregate to generate the final total results.

IV. EXPERIMENT AND DISCUSSION

A. Programming Environment and Experimental Data

The experimental programming environment is as follows:

(1) Computation node: The CPU is Intel core i3 550; and the GPU is NVidia GeForce 410M.

(2) Software environment: The operating system is Microsoft Windows 7; and the program development tools are Visual Studio 2010, MPICH 2.0 and CUDA SDK 4.0;

Using the major road network data set of china as experimental data. As it is shown in figure 9, this data set contains 5240 vertices and 11245 arcs.

B. Experimental Results and Analyzing

To compare the performance of the GPU parallel algorithm, We performed two types of experiments: the comparative experiments between the CPU serial algorithm and the GPU parallel algorithm in a single computing node, and the comparative experiments between different number of computation nodes in the GPU cluster.

Figure 9. The major road network data set of china.

In the comparative experiments between the CPU serial algorithm and the GPU parallel algorithm in a single computing node, we chose 6 different matrix size data, which are 256×256, 512×512, 768×768, 1024×1024, 2048×2048, 3072×3072 and 5240×5240. Perform the

shortest paths computing by respectively using the CPU serial algorithm and the GPU parallel algorithm, and record the time-consuming. The result is shown as table 1.

From table 1, it can be seen that the speed of the GPU-accelerated parallel algorithm is far more than faster than the conventional CPU algorithm. When the size of matrix is small, the speedup is relatively low, because the powerful parallelism computing performance of the GPU can't be took full advantage of, as there is communication cost between device memory and host memory. With the size of matrix increasing, the speedup is also continuously improving; and it can reach about 70 times eventually. It indicates that GPU is especially suitable for these high parallelism and computation-intensive applications.

TABLE I. THE TIME-CONSUMING OF CPU AND GPU FLOYD-WARSHALL SHORTEST PATH ALGORITHM IN A SINGLE COMPUTATION SINGLE NODE

Size of matrix	CPU serial algorithm (s)	GPU parallel algorithm (s)	speedup
256×256	0.269	0.192	1.4
512×512	1.959	0.248	7.9
768×768	6.577	0.261	25.2
1024×1024	15.805	0.352	44.9
2048×2048	114.761	1.782	64.4
3072×3072	390.250	5.512	70.8
5240×5240	878.445	12.150	72.3

In the comparative experiments between different number of computation nodes in the GPU cluster, to better compare the acceleration performance of different number of computation nodes, we conducted 4 experiments, which the number of computation nodes respectively are 1, 4, 9 and 16. Table 2 shows the time-consuming, parallel speedup and parallel efficiency of experiments.

TABLE II. THE EFFICIENCY OF FLOYD-WARSHALL SHORTEST PATH ALGORITHM IN DIFFERENT NUMBER OF COMPUTATION NODES.

Node number	Time-consuming (S)	Parallel speedup	Parallel efficiency	Total speedup
1	12.15	1	1	72.3
4	4.16	2.92	0.73	211.3
9	2.11	5.76	0.64	416.5
16	1.49	8.16	0.51	589.9

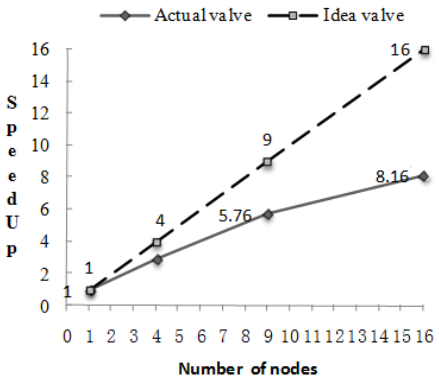

Figure 10. The relationship between the number of computation nodes and parallel speedup.

From table 2, it can be seen, due to the acceleration of MPI + CUDA two-level parallel computing, the speed of the Floyd-Warshall shortest path algorithm was greatly increased. With the increase of the number of the computation nodes, its total speedup continuous improvement and can reach hundreds of time speedup. However, the cluster parallel speedup is not linear growth with the number of computation nodes, as shown in figure 10 and figure 11, the growth of speedup slow down with the number of computation increasing, and the parallel efficiency is also declined at the same time. It is because that with the increase of the number of the computation nodes, the communication costs between nodes is also increasing, thus affecting the computation efficiency.

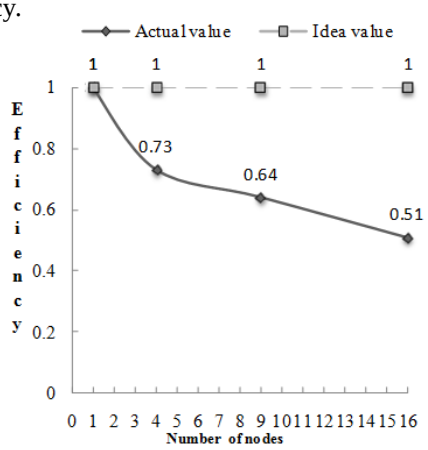

Figure 11. The relationship between the number of computation nodes and parallel efficiency.

V. CONCLUSIONS

The modern GPU is not only a powerful graphics engine but also a highly parallel programmable processor, which can provide hundreds of computing cores to run thousands of threads. Using the GPU's powerful parallel computing performance to the speed up the complex applications has become a more and more important research direction in high-performance computing. NVIDIA's CUDA programming model allows developers to directly use the C/C++ language GPU kernel functions. NVIDIA's CUDA programming model provides a powerful programming environment and instruction set; it allows the programmer to develop GPU kernel functions with common high-level language C/C++ directly. It means that the programmer can focus on the parallel algorithm designing itself, without having to know the GPU hardware programming details. All these advantages of CUDA reduce the GPU programming difficulty and accelerate the popularity of GPU applications. To use the GPU cluster's powerful parallel computing performance, in this paper, we present an all-pairs Shortest paths algorithm using MPI+CUDA hybrid programming model. This parallel algorithm can combine the advantages of MPI and CUDA, and can provide two-level parallel computing. The experimental results show that the MPI+CUDA-based parallel algorithm can take full advantage of the powerful computing capability of the GPU cluster, and can achieve about hundreds of time speedup; The whole algorithm has good computing performance, reliability and scalability, and it is able to

meet the demand of real-time processing of massive spatial shortest path problem.

ACKNOWLEDGMENTS

This work was supported by Hunan Science and Technology Program of China(No.2011SK3130), Zhejiang Provincial Natural Science Foundation of China (No.LQ12D01001), Ningbo City Natural Science Foundation of China(No.2012A610043), Anhui provincial physical geography and human geography key disciplines foundation (No. Asdg1103), National Natural Science Foundation of China(No.41271516).

REFERENCES

[1] Dijkstra Edsger W, "A note on two problems in connexion with graphs, " *Numerische mathematik*, vol. 1, no. 1, pp. 269-271, 1959.

[2] Meyer Ulrich, Sanders Peter, "Δ-stepping: A parallelizable shortest path algorithm, " *Journal of Algorithms*, vol. 49, no. 1, pp. 114-152, 2003.

[3] Brodal Gerth Stølting, Träff Jesper Larsson, Zaroliagis Christos D, "A parallel priority queue with constant time operations", *Journal of Parallel and Distributed Computing*, vol. 49, no. 1, pp. 4-21, 1998.

[4] Madduri Kamesh, Bader David A, Berry Jonathan W, etc, "An experimental study of a parallel shortest path algorithm for solving large-scale graph instances, " *in Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007.

[5] Owens, J. D. Houston, M. Luebke, ect, "GPU computing, " *Proceedings of the IEEE*, Vol. 96, No. 5, pp. 879-899, 2008.

[6] D. Blythe, "Rise of the graphics processor, " *Proceedings of the IEEE*, Vol. 96, No. 5, pp. 761-778, 2008.

[7] Kindratenko Volodymyr V, Enos Jeremy J, Shi Guochun, etc. "Gpu clusters for high-performance computing, " in Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on. pp. 1-8., 2009.

[8] Zhang Yongpeng, Mueller Frank, Cui Xiaohui, etc, "Data-intensive document clustering on graphics processing unit (gpu) clusters, " Journal of Parallel and Distributed Computing, vol. 71, no, 2, pp. 211-224.

[9] Komatitsch Dimitri, Erlebacher Gordon, Göddeke Dominik, etc, "High-order finite-element seismic wave propagation modeling with mpi on a large gpu cluster, " Journal of Computational Physics, vol. 229, no. 20, pp. 7692-7714, 2010.

[10] Xin Yang, Duan-qing Xu, and Lei Zhao, "Incoherent Ray Tracing on GPU, " *Journal of Multimedia*, VOL. 5, NO. 3, PP. 259-267, 2010.

[11] W. Gropp, et al., "A high-performance, portable implementation of the MPI message passing interface standard, " *Parallel Computing*, vol. 22, pp. 789-828, 1996.

[12] P. S. Pacheco, Parallel programming with MPI*:* Morgan Kaufmann, 1997.

[13] Xiaowen Chen, Shuming Chen, Zhonghai Lu, and Axel Jantsch, "Hybrid Distributed Shared Memory Space in Multi-core Processors, " *Journal of* Software, VOL. 6, NO. 12, pp. 2369-2378, 2011.

[14] Nickolls J., Buck I., Garland M., etc, "Scalable parallel programming with cuda", *Queue*, Vol. 6, No. 2, pp. 40-53, 2008.

[15] Owens, J. D. Houston, M. Luebke, D. Green, S. Stone, J. E. Phillips, J. C, "GPU computing, " *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879-899, 2008.

[16] Zuo Chen, Jialiang Ji and Renfa Li, "Asynchronous Parallel Computing Model of Global Motion Estimation with CUDA ", *Journal of computers*, vol. 7, No. 2, pp. 341-348, 2013.

[17] L. Li and A. Malony, "Model-based performance diagnosis of master-worker parallel computations, " Euro-Par 2006 Parallel Processing, pp. 35-46, 2006.

[18] Jing Zhang, Gongqing Wu, Xuegang Hu, Shiying Li and Shuilong Hao, "A Parallel Clustering Algorithm with MPI–MKmeans", *Journal of Computers*, vol. 8, No. 1, pp. 10-17, 2013.

[19] Kirk D., "Nvidia cuda software and gpu parallel computing architecture", in *Proceedings of the 6th International Symposium on Memory Management*, ACM, pp. 103-104, 2007.

[20] J. Katz and J. T. Kider Jr, "All-pairs shortest-paths for large graphs on the GPU, " in *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, pp. 47-55, *2008*.

Wu Qingshuang received the doctor's degree in cartography & geographic information system from Wuhan University, Wuhan, China in 2012. Currently, he is an associate professor at Anhui Normal University, Wuhu, China. His main research interests are parallel computing and spatial analysis.

Tong Chunya received the PhD degree in Photogrammetry and Remote Sensing from Wuhan University, Wuhan, China in 2011. His main interests remote sensing image processing.