Future Generation Computer Systems 123 (2021) 35–47

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Efficient parallel A* search on multi-GPU system

Xin He, Yapeng Yao, Zhiwen Chen, Jianhua Sun, Hao Chen *

College of Computer Science and Electronic Engineering, Hunan University, Changsha, China

ARTICLE INFO

Article history:
Received 3 January 2021
Received in revised form 22 March 2021
Accepted 21 April 2021
Available online 23 April 2021

Keywords:
A* search
GPU
Multi-GPU
Graph partition
Parallelism

ABSTRACT

A* search is a best-first search algorithm that is widely used in pathfinding and graph traversal. To meet the ever-increasing demand of performance, various high-performance architectures (e.g., multi-core CPU and GPU) have been explored to accelerate the A* search. However, the current GPU based A* search approaches are merely designed based on single-GPU architecture. Nowadays, the amount of data grows at an exponential rate, making it inefficient or even infeasible for the current A* to process the data sets entirely on a single GPU.

In this paper, we propose DA*, a parallel A* search algorithm based on the multi-GPU architecture. DA* enables the efficient acceleration of the A* algorithm using multiple GPUs with effective graph partitioning and data communication strategies. To make the most of the parallelism of multi-GPU architecture, in the state extension phase, we adopt the method of multiple priority queues for the *open* list, which allows multiple states being calculated in parallel. In addition, we use the parallel hashing of replacement and frontier search mechanism to address node duplication detection and memory bottlenecks respectively. The evaluation shows that DA* is effective and efficient in accelerating A* based computational tasks on the multi-GPU system. Compared to the state-of-the-art A* search algorithm based on a single GPU, our algorithm can achieve up to 3× performance speedup with four GPUs.

1. Introduction

A* search is a best-first search algorithm, guided by a heuristic function [1], that searches for the shortest path between the initial and the final state. It is widely used in pathfinding and graph traversal. The traditional A* is a sequential algorithm implemented on the CPU. Recently, some researchers have explored using GPU to accelerate the A* algorithm as it offers massively parallel computation for data-parallel problems as well as achieved impressive results [2,3]. However, their algorithm designs only work well in the single-GPU environment without addressing the collaboration between multiple GPUs, thus being incapable of leveraging multiple GPUs for A* search acceleration. Given the compute capability and on-chip memory of a single GPU is rather limited, it is inefficient and even infeasible to process large-scale data on a single GPU. Therefore, it is imperative to propose a new A* search algorithm to address the collaboration issue between multiple GPUs to enable accelerating A* search on the multi-GPU architecture.

To satisfy the rapidly growing demand for computing power in the field of high-performance computing and machine learning [4–7], deploying a system equipped with multiple GPUs for performance acceleration has become prevalent. Fig. 1 shows a typical multi-GPU system. Each node includes a CPU and several GPU devices interconnected by a PCI-e interface or NVLink. Each GPU can directly access its large device memory, much smaller but faster shared memory, and a small pinned area of the host node's DRAM, called zero-copy memory. Therefore, like the A* search algorithm, it needs to extract and store the state from the open queue and the closed queue. Through the collaboration between multiple GPUs, the A* search algorithm can utilize more queues to facilitate the parallel computation among multiple GPUs, which enables the parallelization of the expansion procedure of A* search, thus improving the A* search efficiency as a whole.

Due to the architectural differences, the data placement and transfer strategies for the multi-CPU system are not applicable for multi-GPU architecture. How to design a efficient parallel A* search algorithm in the heterogeneous and complex memory hierarchy of multi-GPU systems remains a daunting task. The existing CPU-based approaches store the graph data in the DRAM and optimize data transfer between devices. In contrast, in multi-GPU architecture, we need to consider how to distribute the graph data among the GPU device memory, GPU shared memory, zero-copy memory, and DRAM on the compute node. More importantly, to optimize the performance on the multi-GPU architecture, it is critical to exploit the data locality in such

* Corresponding author.
E-mail address: haochen@hnu.edu.cn (H. Chen).

https://doi.org/10.1016/j.future.2021.04.011
0167-739X/

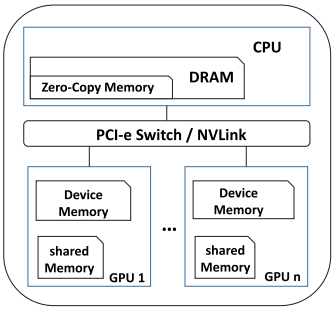


Fig. 1. Multi-GPU node architecture.

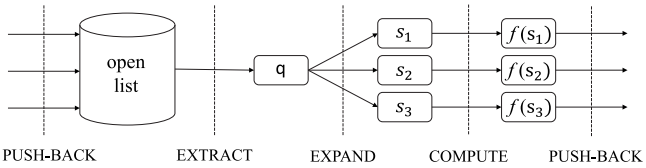


Fig. 2. The workflow of the sequential A* algorithm.

hierarchical memory architectures. Besides, due to the inability of coordinating the computation of multiple GPUs, the existing A* search algorithm based on a single GPU is impractical to effectively exploit multiple GPUs for acceleration in a multi-GPU environment.

In a multi-GPU system, the system includes not only CPU, but also multiple GPU devices. With so many compute devices, we need to consider how to coordinate the computation between compute devices. Specifically, we mainly consider load balancing and communication between compute devices. The load balancing ensures that the compute devices perform the computation in parallel. The key to the communication issue lies in effectively reducing duplicate data movement between devices. As such, how to partition the graph data becomes particularly important.

In this paper, we present a parallel A* search algorithm based on multi-GPU architecture, which can coordinate multiple GPUs for efficient acceleration. With effective graph partitioning and data communication strategies, the entire graph data is distributed onto the DRAM and GPU memory of the associated compute devices to achieve good load balancing and minimize data movement between devices. For the A* search algorithm, the computational overhead of the heuristic function is the main dominating factor that impacts its search performance. In order to reduce this overhead as well as fully exploit the parallelism of multiple GPUs, in the state extension phase, we leverage multiple priority queues for the *open* list, which enables multiple states being calculated in parallel. Besides, to avoid the potential repeated visits to the states of A*, we adopt the parallel hashing of replacement for node duplication. Furthermore, to remedy the memory bottlenecks caused by either the exponentially-growing search space in some A* search tasks or the rapidly expanding *closed* list, we use the frontier search mechanism. In summary, this paper makes the following contributions:

- We present DA*, a parallel A* search algorithm for the multi-GPU architecture, which can effectively exploit the memory bandwidth and compute capabilities of multiple GPUs for acceleration;
- We propose several graph partitioning strategies for different graph types, which enables well-balanced workload among multiple GPUs. These strategies make each compute device process a substantially consistent amount of data, and efficiently use compute resources;
- We present several data communication methods, which can achieve efficient communication between GPU devices and then facilitate the data transfer between calculation results;
- We conduct a comprehensive evaluation of our proposed algorithm on GPU clusters using three representative A* search based applications, the evaluation demonstrates the effectiveness and efficiency of our algorithm. Compared to the state-of-the-art A* algorithms on single GPU, DA* achieves a nearly 3× performance speedup with four GPUs in use on the multi-GPU architecture.

2. Background and motivation

2.1. Traditional A* search algorithm

The traditional A* is a sequential algorithm initially implemented on CPU. Fig. 2 illustrates the workflow of the traditional A* algorithm. The workflow consists of four steps including EXTRACT, EXPAND, COMPUTE and PUSH-BACK.

The EXTRACT operation represents extracting the appropriate node q from the Priority Queue; EXPAND operations are used to expand adjacent nodes of Q; COMPUTE operation is used to compute the heuristic function values of the adjacent nodes of the already expanded nodes; PUSH-BACK operation is to place the computed nodes in the Priority Queue. The algorithm needs to execute these steps iteratively until the extracted state reaches the target state.

To store the search path and related states, the A* search algorithm employs two lists to store the states during the expansion phase, i.e., the *open* list and the *closed* list. The *closed* list stores all the visited states in order to avoid the unnecessary repeated expansion of the same state. This list is often implemented with a linked hash table to detect the duplicated nodes. The *open* list usually stores the successors of the states whose successors have not been fully explored yet. The *open* list uses a priority queue as its data structure, typically implemented by a binary heap. States in the *open* list are sorted according to a heuristic function $f(x)$:

$$f(x) = g(x) + h(x) \tag{1}$$

where the function $g(x)$ is the distance or cost from the starting node to current state x, and the function $h(x)$ defines the estimated distance or cost from current state x to the end node. We call the function value of $f(x)$ the f value. If the f values of a given problem are small integers, the *open* list can also be efficiently implemented with buckets [8].

In each cycle of A* search, the algorithm extracts the node with the minimum f value from the *open* list, expands the adjacency node of the node, and performs duplication detection. Then, it computes the heuristic function values for the corresponding expanded nodes and pushes them back into the *open* list. If some of the newly stored nodes already exist in the *open* list, the f value of the corresponding nodes in the *open* list can only be updated.

To ensure the optimal path can be found, the heuristic function of the A* search algorithm needs to be optimal, which requires that $h(x)$ should not be greater than the actual cost or distance from the give node to the target node. If the search graph is not a tree, a stronger condition called consistency: $h(x) \leq d(x, y)$, where $d(x; y)$ represents the cost or distance from x to y, guaranteeing that once a state is extracted from the queue, the path that it follows is optimal [9].

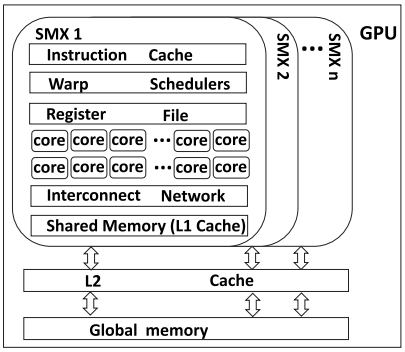


Fig. 3. The architecture of NVIDIA CUDA GPU.

2.2. The architecture of GPU

As a prevalent high-performance architecture, GPU has been widely used in various areas for different purposes such as accelerating big-data processing [10–13] and assisting operating systems [14–19] as a buffer cache. Fig. 3 shows a typical NVIDIA CUDA GPU architecture composed of a set of streaming multiprocessors (SMs) and a GPU main memory (shared among all SMs). Each SM consists of a warp scheduler, arithmetic units, registers, cache, and shared memory. The cores within each SM executes the operation in a SIMD (Single Instruction, Multiple Data) way. But threads in different SMs could run asynchronously because each SM runs independently.

In a GPU device, the smallest unit of execution is the thread. A block represents a group of threads in the same SM. A grid represents the block set which runs the same function. Groups of threads with consecutive thread indexes are bundled into warps. At runtime, a thread block is divided into a number of warps for execution on the cores of an SM. The threads in the warp execute the instruction in a SIMD-like fashion (a program counter will be shared by the threads within a warp). If the execution paths of threads within the same warp are different (e.g., IF-THEN-ELSE branches), a warp divergence will occur. The threads within a block have the same shared memory in an SM, and access to shared memory is much faster than access to the GPU global memory shared by all SMs [20].

2.3. Parallel A* search algorithm

Due to the use of the sequential structure in the traditional A* algorithm, computing the heuristic functions has become the major bottleneck of search efficiency. Particularly, in some A* search based applications, the overhead of computing heuristic functions is costly. Because of the massive parallelism of the GPU, some research works have explored leveraging GPU to accelerate the A* algorithm. The GPU, featured with thousands of CUDA cores, can be a good fit for parallelizing the computation of heuristic functions, such as the calculation of the f values of states in the *open* list. The rationale behind this is based on the observation that the computation of heuristic functions for each expanded state is mutually independent. [2] introduced a parallel method of A* algorithm via parallelizing the calculation of heuristic function and the state expansion process of *open* list. Wang et al. [21] proposed Gunrock, a high-performance graph processing library on the GPU, which also achieved the GPU acceleration of the A* algorithm by utilizing the GPU to parallelize the computational process of the heuristic functions.

How to improve the parallelism of the A* algorithm on GPU faces two challenges. First, in some applications, such as in the search of the grid graph, the degree between the nodes is usually less than 10, which severely hinders the utilization of GPU's parallelism with thousands of cores. Second, as shown in Fig. 2, the algorithm itself contains many sequential execution parts (e.g., the EXTRACT and PUSH-BACK operations), which makes the A* algorithm difficult to compute in parallel.

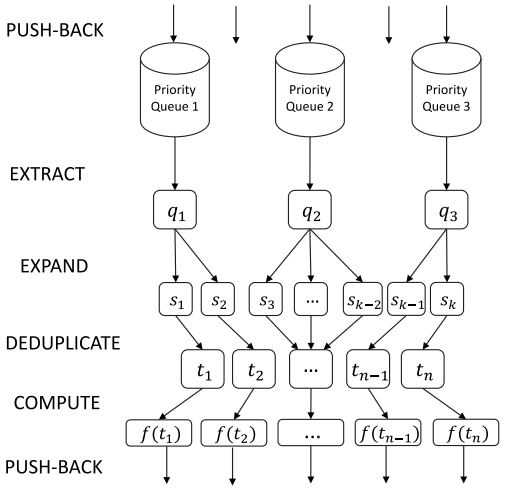


Fig. 4. The workflow of the parallel A* algorithm based on a single GPU.

To overcome the first challenge, we may extract multiple states from the priority queue to enhance the parallelism for the heuristic function. However, the priority queue operations still run in a sequential way. To overcome the second challenge, we may use a concurrent data structure for the priority queue. But the existing lock-free concurrent priority queue cannot run efficiently on the SIMD architecture of a modern GPU processor as it involves the usage of compare-and-swap (CAS) operations [22]. Based on the above methodology, we will introduce an efficient parallel A* algorithm.

Fig. 4 shows the workflow of the parallel A* algorithm based on a single GPU. A large number of priority queues (usually thousands of, for a typical GPU processor) are constructed in the A* search, instead of just one priority queue. In each round, the algorithm could extract multiple states from the multiple priority queues, which enables the parallelization of the sequential execution of the original algorithm. Besides, in the expansion phase, the algorithm increases the number of extended states, which further improves the degree of parallelism for the calculation of heuristic function. As the states are coming from different priority queues, the expanded states may have some duplicate states. The algorithm will detect the duplicate nodes and then remove the duplicate nodes. Finally, the calculated states from the same parent are inserted into different priority queues because states with the same parent tend to have similar priority values. This process is the main execution flow of the parallel A* algorithm, which can take full advantage of the GPU.

2.4. Motivation

The current A* algorithms are mainly designed based on a single CPU, multi-CPU, or a single GPU architectures. Zhou et al. [2] proposed, for the first time, a parallel A* algorithm based on single-GPU architecture and achieved a considerable performance improvement over the sequential A* algorithm on CPU.

However, there are two shortcomings with the existing single-GPU based A* search algorithm. First, the compute resources and on-chip memory of a single GPU is relatively limited. When the amount of data cannot fit in the device memory, it is very

likely that out of memory error would occur. Second, the main way to improve the degree of parallelism is to increase the number of priority queues. When the number of priority queues reaches a certain degree, the algorithm will inevitably encounter corresponding computing bottlenecks.

These shortcomings motivate us to take the initiative to explore the A* algorithm on the multi-GPU architecture. Compared to the single-GPU architecture, there are several unique challenges on the multi-GPU architecture:

1. As shown in Fig. 1, the multi-GPU compute nodes are featured with multi-level memory architecture. We need to consider how to make full use of the hierarchical memory architecture.
2. As the entire data is divided into multiple partitions, we should consider how to solve the problem of data communication and cooperation between GPU devices.
3. How to achieve effective graph partitioning and solve the load balancing problem among GPU devices.
4. As shown in Fig. 4, the algorithm can generate many duplicate nodes in expansion phase. We need to adopt an appropriate method to avoid the duplication to improve search efficiency.
5. Since the A* search often encounters memory bottlenecks when a large number of visited states needs to be stored in a *closed* list, we should consider how to mitigate such bottlenecks.

3. Key issues in parallel A* algorithm design on multi-GPU systems

Given the architectural difference of multi-GPU systems, to adapt the previous CPU/GPU-based A* algorithm to the multi-GPU architecture, we identify four fundamental issues that any practical A* algorithm should consider. These issues are critical in achieving good performance for A* search on the multi-GPU architecture.

Issue 1. How to partition the graph. To fully leverage the parallelism of the multi-GPU architecture, it is necessary to consider how to partition the whole graph data among multiple GPU devices. A good partitioning strategy helps achieve a better load balancing and thus improve the execution efficiency of the algorithm. The differences in the data structure of different graph types lead to different choices of partitioning strategies. For example, due to the characteristics of the grid graph, the number of nodes is proportional to the number of edges, we partition the graph data by nodes. The partitioning method will assign a roughly equal number of nodes and edges to each partition.

Issue 2. How to achieve efficient data communication between the compute devices. The algorithm will select an update set, which contains some common nodes between the partitions. The data of each partition will be directly updated to the update set without direct interaction between partitions. To reduce direct communication between GPU devices, the update set will be placed in zero-memory. In the following sections, we will introduce more details about the communication issue.

Issue 3. How to implement an efficient computing method. First, to take full advantage of the multi-GPU parallelism, the algorithm adopts multiple priority queues instead of a single one. By increasing the number of priority queues, the algorithm not only further expand more adjacent states but also parallelize the computing process of heuristic function on multiple GPU devices, thus speeding up the execution of the algorithm. Second, unlike the priority search algorithm, the A* algorithm needs to compute the corresponding heuristic function in each partition. The value of $h(x)$ in each partition indicates the distance or cost from the current node x to the target node. However, in the graph partition, the distance cost between intermediate nodes needs to be calculated in some cases. It is necessary to consider which node should be chosen as the target node when calculating $h(x)$. If the intermediate node is chosen as the target node, the target node would be frequently changed due to the variation of the intermediate node, which can lead to the increase of computation and wrong results. In contrast, the end node is the final destination and has a fixed position. Therefore, we choose the end node as the target node of each partition. Third, the parallel expansion process could have duplicate states, which requires us to consider how to de-duplicate efficiently.

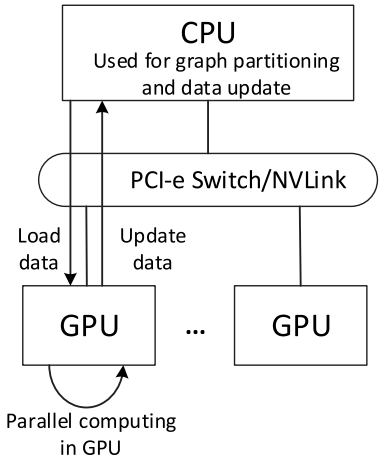


Fig. 5. The workflow of the multi-GPU system for DA*.

Issue 4. How to address the GPU memory bottlenecks. Compared to CPU, the GPU device memory is rather limited. There are two scenarios where the GPU memory bottlenecks can easily occur. First, for A* search, during the expansion process, the algorithm may extend a large number of nodes and a large number of visited states needs to be stored in a *closed* list, which can exhaust large amount of memory. Moreover, the GPU memory bottleneck can be caused by some A* search tasks like sliding puzzle which has a exponentially-growing search space.

4. Proposed algorithm

4.1. Overview

To holistically address the aforementioned four issues, we propose a new parallel A* search algorithm for multi-GPU architecture. Fig. 5 illustrates the main workflow of the proposed algorithm based on the multi-GPU architecture. The CPU is responsible for the partitioning of the graph data and data update. We use different graph partitioning strategies for different graph types and adopt the corresponding communication method based on the chosen graph partitioning strategy (Sections 4.2 and 4.3). In addition, We adapt the execution flow of A* algorithm for multi-GPU architecture using a set of novel techniques including multiple priority queues, parallel hashing of replacement and frontier search mechanism (Section 4.4).

Specifically, the algorithm selects an updated node-set, which is used to perform data update between the compute devices. After partitioning, the partition data will be assigned to the corresponding GPU for computation. Leveraging the parallel A* algorithm, each GPU device will process roughly equal data. As mentioned before, multiple priority queues are used to improve the parallelism of the algorithm, thereby improving the performance of the A* search. The corresponding result calculated on

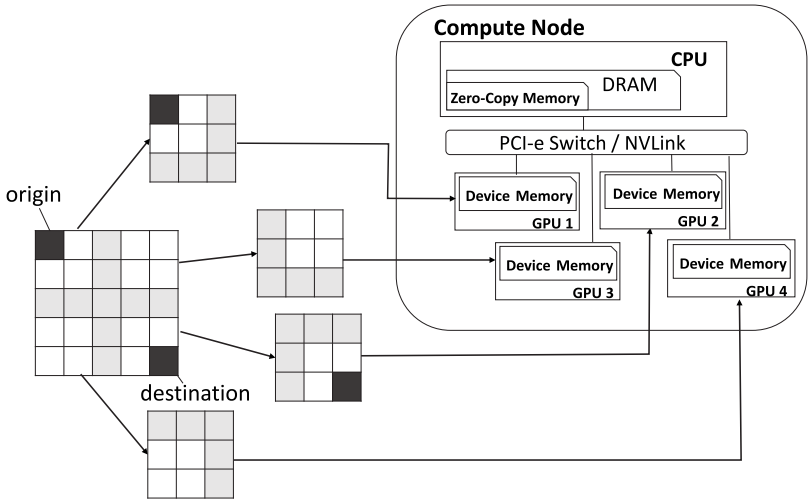


Fig. 6. A grid graph partitioning strategy for 8-connected graph.

the GPUs will be used to update node-set assisting in the communication between the GPU devices (Section 4.3). The GPUs and the CPU are interconnected through the PCI-e bus. We adopt CUDA programming language and Legion programming model [23] for the multi-GPU architecture.

4.2. Graph partitioning strategy

The limited memory of a single GPU severely hinders the data scale that the algorithm can process on the single-GPU architecture. In contrast, the multi-GPU architecture has much more abundant compute and memory resources. There are several different kinds of memory available on the multi-GPU architecture. GPU device memory size is the largest (up to 24 GB on current GPUs). GPU shared memory is much smaller but substantially faster than GPU device memory (up to 96 kB on current GPUs). Zero-copy memory is slower but relatively larger than the GPU device and shared memory. Hence, on the multi-GPU architecture, it is very important to partition the graph data based on the multi-level memory architecture. Though there are several well-known graph partitioning methods, which is very suitable for other best-first search algorithms but difficult to directly use the methods for A* because of its distinct execution characteristics. In the paper, we propose several new methods for partitioning the graph based on the multi-GPU architecture.

4.2.1. Grid graph partitioning strategy

Fig. 6 shows how a grid graph partitioning strategy works for 8-connected graphs, which is commonly used in A* search. The partitioning strategy can partition the grid graph directly since the number of nodes is proportional to the number of edges. Based on the number of GPUs, the graph data will be partitioned as evenly as possible to make the number of nodes in each partition equal. In Fig. 6, the black squares represent starting node and end node respectively, and the gray squares represent the intermediate nodes which are important to the algorithm execution. It works as a bridge for different partitions and will be used for data communication between the partitions. It is also noteworthy that the selection of the intermediate nodes can be different. Generally, depending on the partitioning method, the intermediate node should always be the node where the partition edge locates.

When the size of the graph exceeds the entire GPU memory capacity, the algorithm will try to allocate as much data as possible to each GPU device memory. And the remaining data will be placed into the shared zero-copy memory of the compute node. After placing the data of the graph partitions into the device memory of the associated GPUs, a parallel A* algorithm will be used to search the relevant results.

4.2.2. Directed graph and tree structure partitioning strategy

Directed graph. Many existing distributed graph frameworks, such as PowerGraph [24] and GraphX [25], use the vertex-cut partitioning that optimizes for inter-node communication by minimizing the number of edges spanning different partitions. However, the vertex-cut partitioning method is not suitable for our algorithm. First, it costs a lengthy time to do the vertex-cut partitioning, incurring significant overhead for the algorithm. For example, partitioning the Twitter graph of 1.4B edges into 4 partitions can take a couple of hours on our experimental platform (detailed in Section 6.1). Second, because the data which cannot fit in the GPU device memory is shared among GPUs through a node's zero-copy memory, the number of cross-partition edges is not a good estimate of data transfers.

Therefore, we use the edge-cut partitioning method (as shown in Fig. 7) which assigns a roughly equal number of edges to each partition. Each vertex is assigned a unique number between 0 and $|V| - 1$, where V represents the number of vertexes. The partitioning method determines how to partition the vertices into the device memories of multiple GPUs. In our algorithm, each partition includes sequentially numbered vertices, which allows us to determine the partition by its first and last vertex and all the edges which direct to vertices in that partition. The common vertices generated after the edge-cut partitioning will be combined into an update set and placed in zero-copy memory to minimize the direct interaction between compute devices. Compared with undirected graph data, this partitioning method is more suitable for directed graph and helpful to reduce the occurrence of repeated search.

After partitioning, the algorithm broadcasts the partitioning decision to all GPUs, which can then load the graph from a file system concurrently. All GPUs store mutable vertex properties in the shared zero-copy memory, which can be directly loaded by other GPUs on the computing node.

Tree structure. In some A* search based applications such as sliding puzzle and Rubik's Cube problem, the search space is a tree structure that is generated during search space. The distinct feature of the tree structure is that with the number of layers increases, the number of nodes will increase exponentially. As such, improving parallelism is critical for the reduction of search

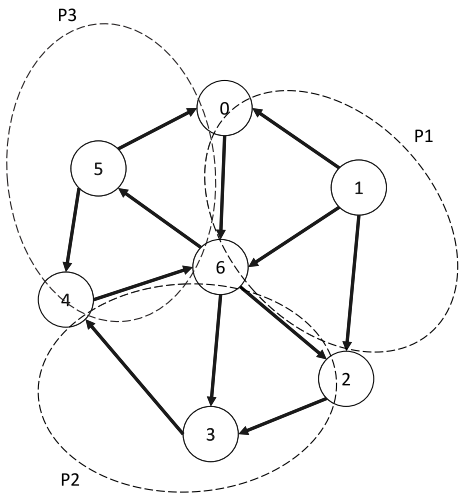


Fig. 7. A edge-cut graph partitioning strategy divides directed graph into 3 GPU devices.

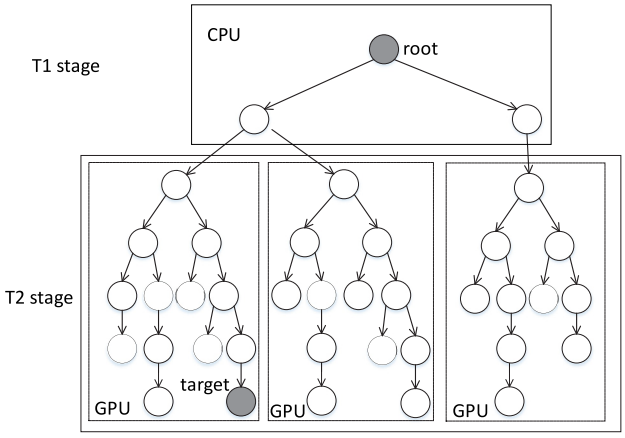


Fig. 8. Tree structure search process.

time. Overall, the first few layers are usually executed on the CPU. In the subsequent calculation, when enough (at least the number of GPUs in practice) states are generated, the algorithm will assign child nodes to the associated GPU devices. As illustrated in Fig. 8, the tree structure is divided into two parts, namely T1 part and T2 part. The algorithm chooses the first two layers as the T1 part since the number of nodes in both the two layers is less than the number of the used GPUs (in this example, we use 3 GPUs). The remaining layers consist of the T2 part. Starting from the root, the algorithm executes the T1 part on the CPU and the T2 part in the GPU device. GPU performs the search process in a top-down manner until the expected target is found.

However, during the search process, it should be noted that the resulting state may appear to be the same as the state in the position the algorithm has already visited. For example, in the 4 puzzles, states have been generated in stage T1, such as 3,1,2,0, and the same state has been generated in stage T2. Therefore, when such a situation occurs, the algorithm terminates the corresponding calculation path to prevent wasting the computing resource of the device.

4.3. Communication between partitions

In this section, we describe how the data communication between GPUs works. As mentioned before, we divide the graph

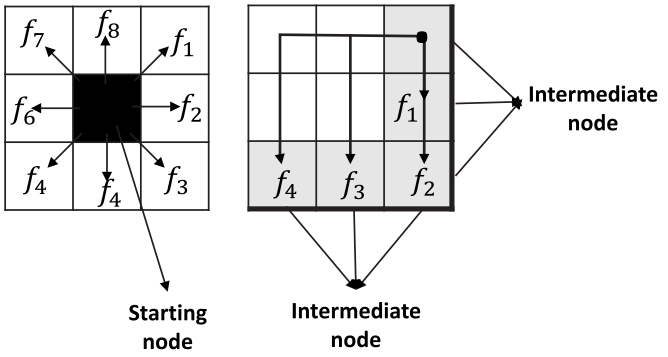


Fig. 9. Search mode within the partition of a grid graph.

evenly into each GPU device, and the intermediate nodes are allocated to the host memory for overall data updates, which helps reduce the communication between the GPU devices and the host memory, thus lowering the communication overhead. Even the communication between the GPU and the host memory relies on a high-speed bus such as NVLink, its transmission speed can still affect the overall performance negatively. On the multi-GPU architecture, the parallel A* algorithm has the data of each graph partition separately calculated on its associated GPU device.

4.3.1. Communication between grid graph partitions

Taking the 8-connected graph as an example for introduction, because the grid graph is an undirected graph where the number of nodes is proportional to the number of edges, when the algorithm partitions the graph, the common nodes between partitions will be used as correlation nodes in adjacent partitions, and the update node set composed of the correlation nodes will be used to update the overall data.

According to the position of the starting node and the target node, there are mainly four cases: (1) if the partition graph does not contain the starting node and the end node, the algorithm will calculate the cost between the intermediate nodes. Actually, it does not mean that every two intermediate nodes have to be calculated, only the cost between intermediate nodes on the search path needs to be calculated, thus not incurring lots of computation on this partition. (2) if the graph partitions only contains the end node, the algorithm will choose to calculate the cost between the end node and the intermediate node. Because when the end node cannot reach each intermediate node, there is no path to reach the end node, and the algorithm will directly terminate the process. (3) if the partition graph contains only the starting node, the algorithm needs to calculate not only the cost from the starting node to the intermediate node but also the cost between the intermediate nodes, because the final path may not only pass by the partition once. (4) when the partition graph contains the starting node and the end node, if the starting node and the end node can be reached directly, the algorithm will terminates the process in GPU devices and then output the result. Otherwise, the algorithm will continue to execute the procedure, according to the above cases.

As shown in Fig. 9, the gray squares represent the intermediate nodes. In the graph partitions, the gray squares reveals the search way of the algorithm within the graph partition. Based on whether the partition graphs include the starting node and the end node, Fig. 9 shows two different search modes, the left figure shows the search mode for the grid graph partition containing the starting node, while the right figure depicts the search mode for the grid graph partition without including the starting node or end node. When the compute devices have calculated the required result, the result will be updated into the memory of

the host synchronously. Then, through the CPU, the results will be used to update the distance between the starting node and each intermediate node. After such process, the algorithm will get the distance cost between the starting node and the end node.

Since the A* search is a random walk algorithm guided by heuristic functions, we need to calculate the corresponding heuristic function. As shown in formula 1, the $h(x)$ value represents the estimated distance or cost from the current node x to the end node. But in the partition graph, we need to calculate the distance cost between the intermediate nodes. For $h(x)$, we still select the estimated distance or cost from the current state x to the end node, because the end node is the final target and have a fixed position. The estimated distance between the intermediate nodes is often calculated. If the algorithm sets some intermediate nodes as the end node, the end node could be changed frequently in the partition graph, which not only increases the amount of calculation but also leads to erroneous final result. Therefore, we do not choose the intermediate nodes as end nodes and instead choose the target node as the fixed end node to compute $h(x)$.

4.3.2. Communication between directed graph partitions and tree structure partitions

In directed graph. In order to transfer the vertex updates between different partitions, we need to select some vertexes as update vertices. The subsequent tasks could use the vertex updates. In the algorithm, the subsequent tasks will utilize the related vertex properties.

As shown in Fig. 7, in computing node, we will use P_i to represent the corresponding partition. The algorithm maintains an updated set (UDS) in each partition, which is used to monitor the common vertices in adjacent partitions. When there is a need to update data between different computing devices, the UDS will play an important role. In INS(P_i), INS represents an in-neighbor set, determines the set of vertices whose properties are used as inputs to process in the partition P_i, ONS(P_i), where ONS represents an out-neighbor set, includes all vertices whose properties are used as outputs to process in the partition P_i. The difference of the two sets are the set of vertices, which are used as input in the partition p_i and whose properties will be used to update between partitions.

$$UDS(P_i) = INS(P_i) - ONS(P_i)$$

At partitioning time, the algorithm obtains the update set by the in-neighbor and out-neighbor sets and places the update set in the host memory. The $\bigcup_{P_i \in N} UDS(P_i)$ represents the union of the update set in each partition. Then the algorithm will use the union update set to schedule the data transfer. At the end of each iteration, each computing device locally collects vertex update data. After completing the previous step, the algorithm sends vertex update data to the union update set, which will be used to obtain the desired result.

In tree structure. The goal is to find out the desired target. As shown in Fig. 8, due to the feature of the tree structure, there is no same child node between each father node. So each GPU device can perform the calculation process independently, resulting in less data communication between partitions. Therefore, it is more important to reasonably partition the graph data.

4.4. Algorithm execution flow

In this section, we will introduce the execution flow of the DA* algorithm in the multi-GPU architecture. Compared with the parallel A* algorithm in the single-GPU environment, our A* algorithm needs to consider not only the parallelization but also collaboration among multiple GPUs. Although many algorithms frameworks, such as GTS [26], Medusa [27], Lux [4], etc., have been proposed based on multi-GPU architectures for search algorithms like depth-first or breadth-first search algorithms. However, the A* search cannot directly leverage these frameworks as the A* is heuristic based search algorithm. Next, we will mainly introduce the execution flow of the A* algorithm on the multi-GPU architecture.

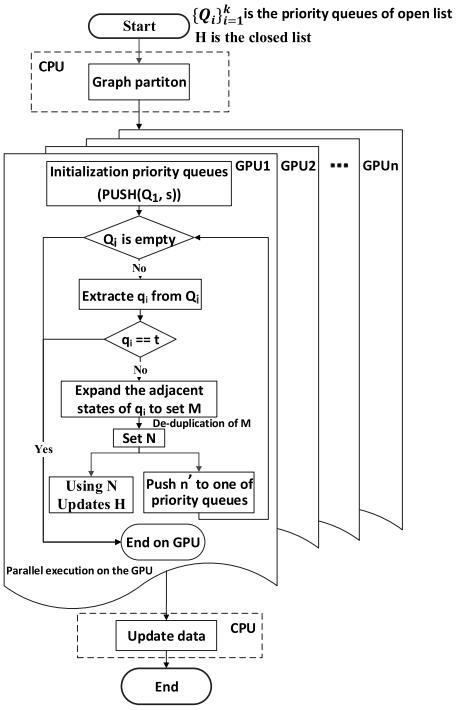


Fig. 10. The main algorithm execution flow of DA* on the multi-GPU system.

Fig. 10 shows the main execution flow of the algorithm. Overall, the CPU is in charge of performing graph partitioning and data update, while the GPU is responsible for the following search process. The algorithm mainly uses multiple priority queues to improve the parallelism of calculating the states on the multi-GPU architecture. Besides, the algorithm selects an update node-set for data update between the compute devices. Based on the corresponding result calculated on the GPUs, it updates the node-set accordingly for the communication between the GPU devices.

At the beginning phase, the algorithm divides the whole graph data into equal partitions based on the number of GPUs available. Specifically, the algorithm will use different partitioning methods based on different graph types as mentioned before. If the graph data cannot fit in the device memory of GPUs, the remaining graph data will be placed in zero-copy memory after graph partitioning. Then, the algorithm will assign graph partitions to the associated GPU devices, and then start executing the corresponding A* search separately on each GPU.

4.4.1. Search procedure on GPU

On GPU devices, the algorithm employs multiple priority queues, such as Q_1, Q_2, and Q_3. First, the starting node is inserted into priority queue Q_1 to initialize the priority queues. Then, the algorithm enters the loop until the target node t is found or all priority queues are empty. Next, the algorithm extracts the nodes from the priority queues Q_i. This operation can extract the minimum state from each priority queue in parallel. If the target node t is found in the extracting state phase, the search process will be terminated and the program will end. Otherwise, the algorithm will continue to perform the next step.

According to the states extracted from the priority queues Q_i, the algorithm expands the adjacent nodes of the states extracted to set M. Because the state extending operations are executed in parallel, which can lead to duplicate states, the algorithm will start de-duplication operation for M. This operation uses a method, called *Parallel Hashing with Replacement*, to execute the corresponding process. The algorithm chooses the states, which are the adjacent states of the element of the set H. Through this step, we can obtain the set N, where the nodes are added to the search path. Next, after the de-duplication, we use the set N to update the nodes in the *closed* list H. If the cost of the state is less than the cost of the same state in the *closed* list H, the algorithm will update the state in the *closed* list H. If the state is in the set N, which is non-existent in the *closed* list, the algorithm will insert the states to the *closed* list H, and priority queues Q_i. Then, the algorithm runs the loop operation. The above process will be terminated once the corresponding end node is found or the priority queues are all empty.

After the processes are completed on GPU devices, the data will be updated into the host memory synchronously. Next, in order to find the shortest path from the starting node to the end node, the algorithm will perform a stitching walk based on the results calculated on GPU. Taking the 8-connected graph as an example, we will set a flag such as 'tag' to 1, in attempt to avoid the repeated access to nodes. Once the node has been visited, the 'tag' will be set to 0. When all successor nodes have been visited during the execution process, the process is terminated and the final results cannot be obtained. Otherwise, the algorithm will continue to run until the target is found.

Algorithm 1: Frontier Search

```
1   procedure frontier_search(now, later)
2    root=start node
3    threshold=root.g
4    insert root into now
5    while now not empty
6         for each node in now
7              if node==target
8                   return target
9              end if
10             if node.f > threshold
11                  insert node into the end of later
12             else
13                  insert children of node to the top of now
14                  remove node from now
15             end if
16        end for
17        tmp=later, later=now, now=tmp
18        increment threshold
19   end while
20   end procedure
```

Fig. 11. The pseudocode of frontier search.

4.4.2. Duplication detection on GPU

In the A* search, we may try to expand a state whose nodes have been already visited. If the f value of the new state is not smaller than the existing state in the *closed* list, it is safe to prevent this state from being visited again. This process is called node duplication detection.

In general, the selection of the method for duplication detection is application dependent. When the amount of graph data can fit in the memory of the GPU, the algorithm could simply use an array to accomplish the duplication detection task. However, the search space of some applications grows exponentially, such as the Rubik's Cube and the sliding puzzle, as shown in Fig. 8. When the search space becomes too large, it is impractical to simply use a pre-allocated table like an array for duplication detection. This is due to the fact that the increasing data size of states stored in the *closed* list can easily exceed the device memory, making it difficult to finish the duplication detection.

A data structure supporting both `INSERT` and `QUERY` operations is a necessity for duplication detection. The `INSERT` operation is to insert a key–value pair to the target data structure. The `QUERY` operation is used to check whether the key exists in the data structure. If so, it returns the associated value of the key. On GPUs, it is a common practice to use a linked hash table or a balanced search tree (e.g., red-black trees) as the data structure. However, it is very difficult to do parallel node duplication detection on GPU using either of them. For instance, it could be troublesome to handle the case in which several different states are inserted into the same position simultaneously in a linked hash table. Hence, to improve the efficiency of node duplication on GPU, it is necessary to find a more suitable data structure for the node duplication detection.

In this paper, we adopt a lightweight probabilistic data structure called `Parallel Hashing with Replacement` for the duplication detection, aiming to simplify the parallel execution in the multi-GPU environments. Multiple hash functions are utilized in `Parallel Hashing with Replacement` to increase the occupancy factor. To handle concurrent access, we employ a lock-free concurrent implementation by leveraging the atomic primitives (particularly atomic-swap), which does not require the use of synchronization operations like locks. As a result, there is no need for multiple hash table instances. The prominent advantage of this method is that a small number of duplicate nodes can be ignored during detection while enabling a fast and simplified implementation. The key to the method lies in the fact that all duplicate nodes do not have to be detected. The rationale is that some states with duplicated nodes in the *open* list have no impact on the correctness of the algorithm. If some nodes are allowed to be ignored, the algorithm does not require strict collision detection. During the replacement phase of the duplication detection, a new node will replace the old one in its position. The old node will be directly discarded instead of being placed in another position. This alternative approach makes the parallelization of duplication detection on the multi-GPU architecture much faster and easier.

4.4.3. GPU memory bounded A* search

When a large number of visited states need to be stored into the *closed* list, the A* search will suffer memory bottlenecks, As the *closed* list is critical for duplication detection and finding out the optimal path, when the device memory is insufficient, it is not possible to continue to run the search process to find the optimal path, which is most likely to happen for the A* based search task (e.g., sliding puzzle) with exponentially-growing search space and thus ends up with program errors.

In the previous A* search algorithm, there are some solutions to this problem. The prevailing methodology is to give up the searching of the optimal path when memory is exhausted. For instance, the memory-bounded bidirectional search [28], the memory-bounded A* Graph Search [29], and the frontier Search [30] all adopts this methodology. Among these three methods, the frontier search is more suitable and efficient to implement on GPUs. Consequently, we leverage the frontier search mechanism to address the memory bottlenecks for our algorithm.

As shown in Figure Fig. 11, for the frontier search, all the nodes in the *closed* list will be discarded and the *closed* list is

not retained such that the memory consumption can be lowered. Though the *open* list remains, the order of elements in the *open* list is no longer maintained. The *now* list and *later* list are used to represent the *open* list, the most promising states based on their f value in the *open* list are saved by setting thresholds and making dynamic updates. Specifically, the f value of the top node in the *now* list is compared with the threshold. If it is greater than the threshold, the node will be moved to the end of the *later* list. Otherwise, its children will be inserted into the top of the *now* list, and the current node will be removed from the *now* list and then continue the traverse. This process maintains the list in a weak sort manner. If no target is found in the *now* list during the traverse of the *now* list, the threshold is incremented while the elements of the *later* list are interchanged with the *now* list, then a new round of traverse starts. Although frontier search needs to use the now and *later* lists to maintain the *open* list, there is no sorting overhead and extra memory overhead. Besides, we use the scan primitive [31] and GPU radix sorting for the selection of states on the GPU.

5. Implementation

We implement DA* by leveraging some techniques introduced in the graph processing framework Lux [4], which is a distributed multi-GPU system for fast graph processing. The advantage of Lux is that it achieves fast graph processing by exploiting the aggregate memory bandwidth of multiple GPUs and taking advantage of locality in the memory hierarchy of multi-GPU clusters. The underlying implementation of Lux adopts a programming model called legion [23]. Legion is a data-centric programming model for developing high-performance applications for distributed heterogeneous architectures, while being compatible with the NVIDIA CUDA programming model. We employ Legion to build a processing framework for a collaborative multi-GPU environment. We adopt CUDA programming language to implement our algorithm, including approximately 1500 lines of code.

6. Evaluation

6.1. Experimental setup

Experimental Platforms. We set up an experimental system which contains two 12-core Intel Xeon E5-2678 CPUs with 64GB system memory, four NVIDIA V100 GPUs. The GPUs are attached to the server through PCIe 3.0. We configure the experiment with the use of different number of GPU devices (i.e., 1, 2, 4 GPUs respectively) for testing and comparison. The operating frequency of the CPU core are running at 2.5 GHz. Each V100 GPU has 16GB of off-chip global memory and 5120 CUDA cores. The operating system is Ubuntu 16.04. We compare the performance of the parallel A* algorithm in a single GPU environment against that of the A* algorithm in a multi-GPU environment. Unless indicated otherwise, GA* represents the A* algorithm in the single GPU environment. DA* represents the A* search algorithm in the multi-GPU environment.

Workloads. We evaluate the performance of our proposed A* search algorithm in three representative problems including sliding puzzle, pathfinding, and protein design. For the first two experiments, we set the number of priority queues to 2496 and 9984 to evaluate the impact of using different number of priority queues. The evaluated metrics we use are execution time, the total number of expansion state, and the expansion number per second.

Datasets. For sliding puzzles, we use 15 puzzles and 24 puzzles for comparison, and the data is randomly generated. For pathfinding, we use an 8-connected grid graph, and the node in the graph has an edge to an adjacent node. The size of the grid graph we use in the experiment is 10 000 $\times$ 10 000. We choose three kinds of data of protein structures for protein design, which are 2CS7, 2DSX, 3D3B respectively.

6.2. Sliding puzzle

Sliding Puzzle has been widely used to evaluate the performance of heuristic search algorithms [32,33]. The heuristics function we choose is the disjoint pattern database [34], which uses the sum of the results from multiple pattern databases [35] and still guarantees consistency. *Pattern databases* are usually used in heuristic search applications, such as solving Rubik's cube [36], and are much more efficient than the *Manhattan distance* based heuristic functions.

Tables 1 and 2 shows the detailed comparison of GA* and DA* on 15-puzzles and 24-puzzles problems respectively. Compared with GA* algorithm based on the single-GPU architecture, DA* can use more memory and compute resources. Due to the smaller amount of data generated by 15-puzzle, we can find that under the same number of GPUs, using different number of priority queues does not have a significant impact on execution time. This is because the search space of 15-puzzle is relatively small and the GPU has relatively strong computing power.

However, in the case of solving 24-puzzles, the amount of data is considerably higher than 15-puzzles. As shown in Table 2, with roughly the same number of execution steps and the same number of priority queues, it can be seen that when the number of GPUs is 2, DA* algorithm can achieve up to 2$\times$ speedup compared with GA* algorithm. When the number of GPUs reaches 4, the performance improvement over GA* algorithm can reach nearly 3$\times$. It should also be noted that when the number of GPUs is the same, the number of priority queues is directly proportional to the execution time in both 15-puzzles and 24-puzzles cases. The only difference is that the impact can be more significant with a larger amount of data.

Figs. 12 and 13 show the execution time of GA* and DA* using different number of GPUs and priority queues on 15-puzzles problems and 24-puzzles problems respectively. Since GA* can only run in the single-GPU environment, we only report its results using one GPU. It can be seen that when the number of GPU is 1, the execution time of DA* and GA* algorithms are almost the same, this is because in the single-GPU environment, the calculation processes of the two algorithms on the GPU are similar and our algorithm does not lead to extra performance overhead than GA*. More importantly, with the increasing of the number of GPUs, DA* is constantly improved in search performance regardless of the number of priority queues. Specifically, it achieves 2.5X higher performance than the GA* when using 4 GPUs. The continuous performance improvements of DA* with the increasing number of GPUs can be attributed to the fact that the DA* achieves the collaboration between multiple GPUs and also can effectively exploit the compute and memory resources of multiple GPUs.

When the number of compute devices continues to increase, the cost of partitioning graph and the communication between computing devices will have non-negligible effect on the total execution time. Therefore, we will notice that when the number of compute devices increases, DA* algorithm does not exhibit a linear performance growth. It should be pointed out that due to the limited memory capacity of a single GPU, In order for the comparison to GA* algorithm, the maximum data size we choose is only 5 $\times$ 5 sliding puzzles.

Table 1
The comparison results of GA* and DA* on 15-puzzles problems (61 steps) with the different number of the parallel priority queues and GPUs. Execution Time is measured in a millisecond, "#of state" is the number of states that the algorithm expands during the computation time, "Exp. Rate" indicates the number of nodes expanded per millisecond, and PPQ indicates the number of the parallel priority queue.

		GA (1GPU)	DA (1GPU)	DA (2GPUs)	DA (4GPUs)
2496 PPQs	Exe. time	183	185	112	76
	#of state	3,295,247	3,245,831	3,390,979	3,631,395
	Exp. Rate	17,909.1	17,913.8	30,276.3	48,478.6
9984 PPQs	Exe. time	172	176	104	70
	#of state	3,135,552	3,166,413	3,375,330	3,784,753
	Exp. Rate	18,020.4	18,409.4	32,455.1	54,067.9

Table 2
The comparison results of GA* and DA* on 24-puzzles problems (64 steps) with the different number of the parallel priority queues and GPUs. Execution Time is measured in a millisecond, "#of state" is the number of states that the algorithm expands during the computation time, "Exp. Rate" indicates the number of nodes expanded per millisecond and PPQ indicates the number of the parallel priority queue.

		GA (1GPU)	DA (1GPU)	DA (2GPUs)	DA (4GPUs)
2496 PPQs	Exe. time	4521	4496	2384	1328
	#of state	73,408,160	74,390,816	73,719,754	69,977,898
	Exp. Rate	16,237.1	16.546.2	32.276.6	52,694,2
9984 PPQs	Exe. time	2170	2191	1056	605
	#of state	67,693,010	67,567,125	66,282,797	66,905,256
	Exp. Rate	31,194.9	30,838.4	58,040.9	92,672.5

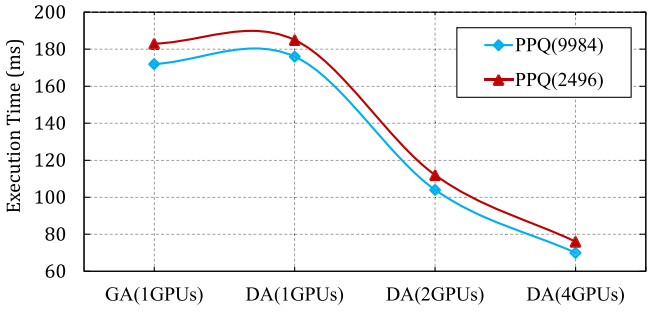


Fig. 12. The execution time of the two algorithms using different number of priority queues and GPUs on 15-puzzles problems.

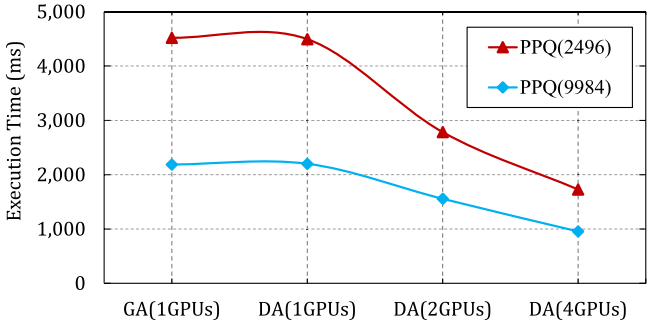


Fig. 13. The execution time of GA* and DA* using different number of priority queues and GPUs on 24-puzzles problems.

6.3. Pathfinding

Pathfinding is an A* search based application typically used to find the optimal path between two nodes in the graph. In our experiments, we evaluate the shortest time of finding the optimal path between two nodes in a grid graph. The heuristic function we choose is the *diagonal distance* instead of the *Manhattan distance*.

In the experiments, we use two pathfinding approaches (i.e., Random and Zigzag) to conduct comparative experiments, and

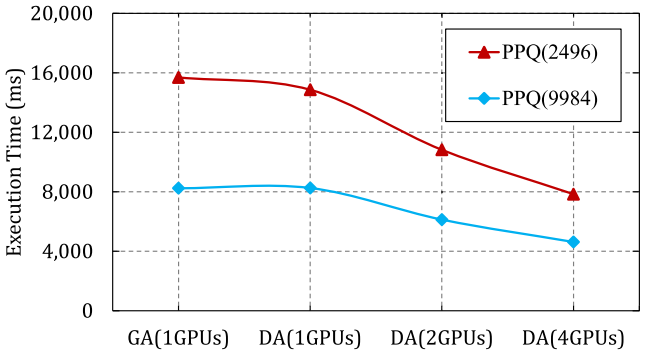


Fig. 14. The execution time of GA* and DA* with different number of priority queues and GPUs (Zigzag).

ensure that the number of steps taken each time is equivalent as to minimize the error. The Random method is that we randomly set a certain number of non-passable points in the grid, and let the algorithm automatically generate the corresponding path. The Zigzag method implies that the path is to be walked in a "Z" shape, and it tends to follow the hypotenuse of the current state.

Tables 3 and 4 show the comparison results between GA* and DA* using zigzag and random pathfinding methods respectively. We find that in the case of the same number of priority queues, no matter Random or Zigzag pathfinding mode, with the increase in the number of GPUs, the performance DA* algorithm is constantly improved. This is because the algorithm can effectively leverage the computing power of multi-GPU architecture for acceleration. Compared with GA* algorithm based on a single GPU architecture, it can expand more nodes per unit time, which is reflected by the increase of Exp. Rate shown in both Tables. As a result, the search efficiency of the DA* algorithm is improved. Besides, it can be observed that when the number of priority queues is specified and the number of GPUs reaches 4, the performance of the DA* algorithm can be improved by up to 3×.

Figs. 14 and 15 show the performance comparison of GA* and DA* using Zigzag and Random pathfinding modes respectively, we observe that as the number of GPUs increases, the performance of DA* algorithm in both cases is constantly improved.

Table 3
The comparison results of GA* and DA* on pathfinding problems (zigzag: 14 239 steps) with the different number of the parallel priority queues and GPUs. Execution Time is measured in a millisecond, "#of state" is the number of states that the algorithm expands during the computation time, "Exp. Rate" indicates the number of nodes expanded per millisecond and PPQ indicates the number of the parallel priority queue.

		GA (1GPU)	DA (1GPU)	DA (2GPUs)	DA (4GPUs)
2496 PPQs	Exe. time	15,690	14,862	8823	5441
	#of state	48,203,231	46,101,924	48,147,383	59,076,021
	Exp. Rate	3072.2	3102.1	5583.6	10,857.2
9984 PPQs	Exe. time	8236	8256	5130	3124
	#of state	74,947,640	75,246,124	78,627,736	86,223,939
	Exp. Rate	9001.6	9114.4	15,623.2	29,094.1

Table 4
The comparison results of GA* and DA* on pathfinding problems (random: 12 251 steps) with the different number of the parallel priority queues and GPUs. Execution Time is measured in a millisecond, "#of state" is the number of states that the algorithm expands during the computation time, "Exp. Rate" indicates the number of nodes expanded per millisecond and PPQ indicates the number of the parallel priority queue.

		GA (1GPU)	DA (1GPU)	DA (2GPUs)	DA (4GPUs)
2496 PPQs	Exe. time	12,496	12,134	7431	4652
	#of state	37,420,672	38,367,708	39,060,988	41,890,201
	Exp. Rate	2994.6	3162.0	5376.6	9004.8
9984 PPQs	Exe. time	9546	9577	5532	3746
	#of state	76,125,411	76,684,862	87,749,060	91,422,094
	Exp. Rate	7974.5	8007.3	15,850.5	27,887.1

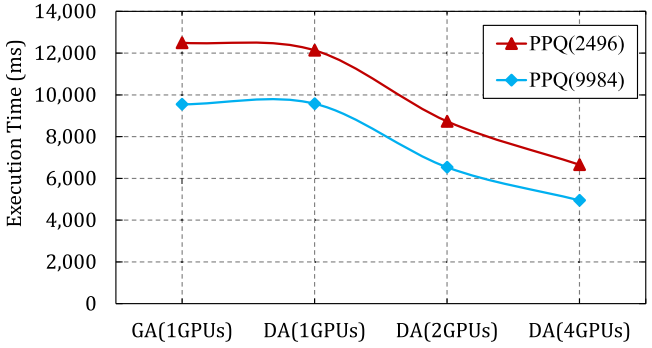


Fig. 15. The execution time of GA* and DA* with different number of priority queues and GPUs (Random).

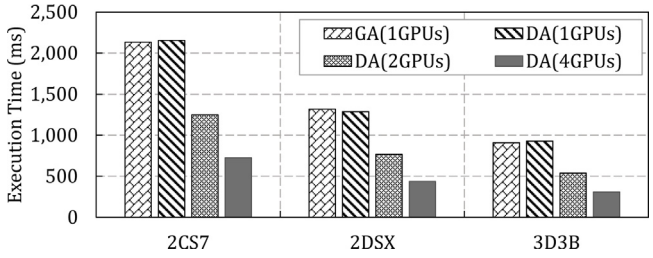


Fig. 16. The execution time of GA* and DA* using different number of GPUs and different kinds of protein data structures on protein design problems.

Nevertheless, the algorithm does not exhibit a linear performance growth with the number of GPUs. Due to the increase of partitions, the graph partitioning process and communication between computing devices also have extra impact on the algorithm execution efficiency. Meanwhile, it will also take more time to find the path from the result data of the partition. In summary, based on the above experiments, we can find that our DA* algorithm can achieve significant performance improvement in a multi-GPU architecture for the pathfinding application.

6.4. Protein design

Protein design is an important problem in computational biology and can be expressed as the most likely explanation for finding a graphical model with a complete graph. Numerous search algorithms have been developed to solve the protein design problem [37–40]. We adopt the energy function introduced in [2] as the heuristic function for DA*.

Fig. 16 shows the performance comparison results between GA* and DA* using different number of GPUs and different kinds of protein data structures on the protein design problems. In this experiment, both GA* and DA* use 4992 priority queues. We evaluate on three kinds of data of protein structures which are 2CS7, 2DSX, 3D3B respectively. In addition, we adopts the A* tree search for this experiment. In A* tree search, since tree nodes are not expanded multiple times, the gap between expanded states in the sequential and parallel versions can be effectively reduced, thus minimizing the impact of repeated expansions on search time. As shown in Fig. 16, for all the three protein structures, the DA* performs as well as the GA* algorithms in the single GPU environment. As more compute devices are involved in the computation, the search efficiency of the DA* is constantly improved while the GA* are unable to use multiple GPUs. When the number of GPUs in use reaches 4, the algorithm can achieve more than 2x performance speedup for all three cases compared to the use of a single GPU.

7. Related work

A* search algorithm based on the CPU architecture. The traditional A* search algorithm is initially proposed and implemented based on CPU architecture, which employs a sequential execution fashion. With the advent of multi-core CPUs, Burns et al. [33] proposed a general approach for best-first on a multi-core system with shared memory. The method uses abstraction to partition the state space and detect duplicate states without requiring frequent locking.

For Frontier A* (FA*) algorithm augmented, Robert et al. [41] presented sequential and parallel algorithms with a form of Delayed Duplicate Detection (DDD). Hansen et al. [42] proposed a novel approach to parallelize graph search using structured duplicate detection. In the parallel graph search, this approach is also

used to reduce the number of slow synchronization operations needed. Kishimoto et al. [43] introduced a distributed A* search algorithm, called HDA*, on a CPU cluster by assigning internal states to different machines, which uses a hash function. Nissim et al. [44] proposed a multi-agent A* for parallel and distributed system, which delivers super-linear speedup on problems where the agents are loosely coupled. However, the above A* algorithms are all focused on CPU systems. Due to the distinct architectural features, the designs used in these CPU based A* algorithms cannot be directly applied to the design of A* on GPU while our work targets the GPU based A* algorithm.

A* search algorithm based on the GPU architecture. Due to the massive hardware parallelism of GPU, many efforts have been devoted to exploring leveraging GPU to accelerate A* search both from academia and industry. Researchers from NVIDIA corporation introduced an efficient GPU implementation for multi-agent A* search, i.e., finding the shortest paths between multiple pairs of nodes in parallel in a sparse graph, based on the CUDA programming environment [3], which is however restricted to pathfinding problem solving. Sulewski et al. [45] exploits the parallel computing power of graphics cards to accelerate the state-space search, utilizing perfect Hashing functions to accelerate state-space exploration.

Hayakawa et al. [46] explored using GPU for the parallelization of Rubik's cube optimal solver. The method is based on Korf's algorithm. Zhou et al. [2] presented A* search algorithm based on GPU, called GA*. The algorithm is the earliest parallel A* search algorithm based on the single-GPU architecture, and it can only processes data efficiently on the single GPU environment without being able to scale to multi-GPUs. To solve Li's modularity clustering problem, Santiago et al. [47] proposed a novel parallel anytime A* for graph and network clustering, but it is too ad hoc to be applied to other A* based problems and also cannot leverage multiple GPUs for acceleration. In contrast, our DA* is more general in the sense that it can be used to solve a wider range of A* based problems. More importantly, our DA* can work in both single-GPU and multi-GPU environments while achieving excellent search performance across numerous computational tasks.

8. Conclusion and future work

In this paper, we propose DG*, a parallel A* algorithm which can coordinate multiple GPUs for accelerating A* search efficiently. DG* employs different partition and communication strategies for different graph types on the multi-GPU architecture. To facilitate the parallel computation on multiple GPUs, we adapt the execution flow of A* algorithm for the multi-GPU architecture by leveraging multiple priority queues. Moreover, we also propose corresponding solutions to address the node duplication detection and GPU memory bottleneck respectively. The evaluation demonstrates that compared to the A* algorithm in a single GPU environment, our algorithm achieves significant performance improvement particularly with processing large-scale data on the multi-GPU architecture.

In terms of future work, we plan to further explore the A* algorithm for multi-GPU and multi-host environment. How to reduce the communication overhead between hosts with the increase of the number of hosts is challenging. In addition, when the increasing of the number of GPUs leads to more partitions, the improvement of performance could be limited. How to address this issue remains to be investigated.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: This research was supported by the National Key Research and Development Program of China (No.2018YFB1003502), and by the National Science Foundation of China under Grants 61772183 and 61972137. The opinion of the paper is only on behalf of the authors and has nothing to do with the funding agencies.

Acknowledgment

This research was supported by the National Key Research and Development Program of China (No. 2018YFB1003502), and by the National Science Foundation of China under Grants 61772183 and 61972137.

References

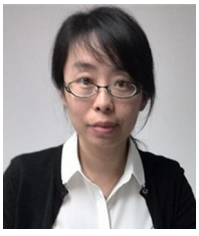
[1] P.E. Hart, N.J. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE Trans. Syst. Sci. Cybern. 4 (2) (1968) 100–107.
[2] Y. Zhou, J. Zeng, Massively parallel a* search on a gpu, in: Twenty-Ninth AAAI Conference on Artificial Intelligence, 2015.
[3] A. Bleiweiss, Gpu accelerated pathfinding, in: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware, Eurographics Association, 2008, pp. 65–74.
[4] Z. Jia, Y. Kwon, G. Shipman, P. McCormick, M. Erez, A. Aiken, A distributed multi-gpu system for fast graph processing, Proc. VLDB Endow. 11 (3) (2017) 297–310.
[5] O. Yadan, K. Adams, Y. Taigman, M. Ranzato, Multi-gpu training of convnets, 2013, arXiv preprint arXiv:1312.5853.
[6] W. Zhong, J. Sun, H. Chen, J. Xiao, Z. Chen, C. Chang, X. Shi, Optimizing graph processing on gpus, IEEE Trans. Parallel Distrib. Syst. 28 (4) (2017) 1149–1162.
[7] J. Sun, H. Chen, L. He, H. Tan, Redundant network traffic elimination with GPU accelerated rabin fingerprinting, IEEE Trans. Parallel Distrib. Syst. 27 (7) (2016) 2130–2142.
[8] R. Dial, F. Glover, D. Karney, D. Klingman, Shortest path forest with topological ordering: An algorithm description in sdl, Transp. Res. B 14 (4) (1980) 343–347.
[9] J. Pearl, Heuristics: Intelligent Search Strategies for Computer Problem Solving, Addison-Wesley Pub. Co., Inc., Reading, MA, 1984.
[10] C. Chen, K. Li, A. Ouyang, Z. Zeng, K. Li, Gflink: An in-memory computing architecture on heterogeneous CPU-gpu clusters for big data, IEEE Trans. Parallel Distrib. Syst. 29 (6) (2018) 1275–1288.
[11] C. Chen, K. Li, A. Ouyang, Z. Tang, K. Li, Gpu-accelerated parallel hierarchical extreme learning machine on flink for big data, IEEE Trans. Syst. Man Cybern.: Syst. 47 (10) (2017) 2740–2753.
[12] C. Chen, K. Li, A. Ouyang, K. Li, Flinkcl: An opencl-based in-memory computing architecture on heterogeneous cpu-gpu clusters for big data, IEEE Trans. Comput. 67 (12) (2018) 1765–1779.
[13] J. Chen, K. Li, Z. Tang, K. Bilal, S. Yu, C. Weng, K. Li, A parallel random forest algorithm for big data in a spark cloud computing environment, IEEE Trans. Parallel Distrib. Syst. 28 (4) (2017) 919–933.
[14] H. Chen, J. Sun, L. He, K. Li, H. Tan, Bag: managing gpu as buffer cache in operating systems, IEEE Trans. Parallel Distrib. Syst. 25 (6) (2014) 1393–1402.
[15] L. Shi, H. Chen, J. Sun, K. Li, Vcuda: gpu-accelerated high-performance computing in virtual machines, IEEE Trans. Comput. 61 (6) (2012) 804–816.
[16] H. Tan, Y. Tan, X. He, K. Li, K. Li, A virtual multi-channel gpu fair scheduling method for virtual machines, IEEE Trans. Parallel Distrib. Syst. 30 (2) (2019) 257–270.
[17] H. Li, K. Li, J. An, K. Li, Msgd: a novel matrix factorization approach for large-scale collaborative filtering recommender systems on gpus, IEEE Trans. Parallel Distrib. Syst. 29 (7) (2018) 1530–1544.
[18] K. Li, W. Yang, K. Li, Performance analysis and optimization for spmv on gpu using probabilistic modeling, IEEE Trans. Parallel Distrib. Syst. 26 (1) (2015) 196–205.
[19] W. Yang, K. Li, Z. Mo, K. Li, Performance optimization using partitioned spmv on gpus and multicore cpus, IEEE Trans. Comput. 64 (9) (2015) 2623–2636.
[20] S. Horie, A. Fukunaga, Block-parallel ida* for gpus, in: Tenth Annual Symposium on Combinatorial Search, 2017.

[21] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, J.D. Owens, Gunrock: A high-performance graph processing library on the GPU, in: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2016, pp. 1–12.
[22] J. Fuentes, W.-y. Chen, G.-y. Lueh, A. Garza, I.D. Scherson, Simd-node transformations for non-blocking data structures, in: International Conference on Parallel Processing and Applied Mathematics, Springer, 2019, pp. 385–395.
[23] M.E. Bauer, Legion: Programming Distributed Heterogeneous Architectures with Logical Regions (Ph.D. thesis), Stanford University, 2014.
[24] J.E. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin, Powergraph: Distributed graph-parallel computation on natural graphs, in: Presented As Part of the 10th {*USENIX*} Symposium on Operating Systems Design and Implementation ({*OSDI*} 12), 2012, pp. 17–30.
[25] J.E. Gonzalez, R.S. Xin, A. Dave, D. Crankshaw, M.J. Franklin, I. Stoica, Graphx: Graph processing in a distributed dataflow framework, in: 11th {*USENIX*} Symposium on Operating Systems Design and Implementation ({*OSDI*} 14), 2014, pp. 599–613.
[26] M.-S. Kim, K. An, H. Park, H. Seo, J. Kim, Gts: A fast and scalable graph processing method based on streaming topology to gpus, in: Proceedings of the 2016 International Conference on Management of Data, ACM, 2016, pp. 447–461.
[27] J. Zhong, B. He, Medusa: Simplified graph processing on gpus, IEEE Trans. Parallel Distrib. Syst. 25 (6) (2013) 1543–1552.
[28] H. Kaindl, A. Khorsand, Memory-bounded bidirectional search, in: AAAI, 1994, pp. 1359–1364.
[29] R. Zhou, E.A. Hansen, Memory-bounded a* graph search, in: FLAIRS Conference, 2002, pp. 203–209.
[30] R.E. Korf, W. Zhang, I. Thayer, H. Hohwald, Frontier search, J. ACM 52 (5) (2005) 715–748.
[31] S. Sengupta, M. Harris, Y. Zhang, J.D. Owens, Scan primitives for gpu computing, in: Graphics Hardware, Vol. 2007, 2007, pp. 97–106.
[32] R.E. Korf, Depth-first iterative-deepening: An optimal admissible tree search, Artif. Intell. 27 (1) (1985) 97–109.
[33] E. Burns, S. Lemons, W. Ruml, R. Zhou, Best-first heuristic search for multicore machines, J. Artificial Intelligence Res. 39 (2010) 689–743.
[34] R.E. Korf, A. Felner, Disjoint pattern database heuristics, Artif. Intell. 134 (1–2) (2002) 9–22.
[35] J.C. Culberson, J. Schaeffer, Pattern databases, Comput. Intell. 14 (3) (1998) 318–334.
[36] R.E. Korf, Finding optimal solutions to rubik's cube using pattern databases, in: AAAI/IAAI, 1997, pp. 700–705.
[37] A.R. Leach, A.P. Lemon, Exploring the conformational space of protein side chains using dead-end elimination and the a* algorithm, Proteins: Struct. Funct. Bioinform. 33 (2) (1998) 227–239.
[38] A.A. Canutescu, A.A. Shelenkov, R.L. Dunbrack Jr, A graph-theory algorithm for rapid protein side-chain prediction, Protein Sci. 12 (9) (2003) 2001–2014.
[39] B.R. Donald, Algorithms in Structural Molecular Biology, MIT Press, 2011.
[40] Y. Zhou, W. Xu, B.R. Donald, J. Zeng, An efficient parallel algorithm for accelerating computational protein design, Bioinformatics 30 (12) (2014) i255–i263.
[41] R. Niewiadomski, J.N. Amaral, R.C. Holte, Sequential and parallel algorithms for frontier a* with delayed duplicate detection, in: AAAI, 2006, pp. 1039–1044.
[42] R. Zhou, E.A. Hansen, Parallel structured duplicate detection, in: AAAI, 2007, pp. 1217–1224.
[43] Y. Kobayashi, A. Kishimoto, O. Watanabe, Evaluations of hash distributed a* in optimal sequence alignment, in: Twenty-Second International Joint Conference on Artificial Intelligence, 2011.
[44] R. Nissim, R.I. Brafman, Multi-agent a* for parallel and distributed systems, in: ICAPS Workshop on Heuristics and Search for Domain-Independent Planning, 2012, pp. 43–51.
[45] D. Sulewski, S. Edelkamp, P. Kissmann, Exploiting the computational power of the graphics card: Optimal state space planning on the gpu, in: Twenty-First International Conference on Automated Planning and Scheduling, 2011.
[46] H. Hayakawa, H. Murao, Optimal rubik's cube solver on gpu, in: GPU Technology Conference, 2013.
[47] R.F.D.S. Mendes, R. De Santiago, L.C. Lamb, Novel parallel anytime a* for graph and network clustering, in: 2018 IEEE Congress on Evolutionary Computation (CEC), IEEE, 2018, pp. 1–6.

Xin He is a Ph.D student at the College of Computer Science and Electronic Engineering, Hunan University, China. His research interests are in parallel computing and multi-core many-core systems.

Yapeng Yao is a master student at the College of Computer Science and Electronic Engineering, Hunan University, China. His research interests are in GPU-based parallel computing.

Zhiwen Chen received the PhD degree from College of Computer Science and Electronic Engineering, Hunan University, China. He is currently a postdoctoral fellow in the College of Computer Science and Electronic Engineering, Hunan University. His research interests are parallel computing and multi-core systems, persistent memory systems, and graph computing systems.



Jianhua Sun is a professor at the College of Computer Science and Electronic Engineering, Hunan University, China. She received the Ph.D degree in Computer Science from Huazhong University of Science and Technology, China in 2005. Her research interests are in security and operating systems. She has published more than 70 papers in journals and conferences, such as IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Computers.

Hao Chen received the BS degree in chemical engineering from Sichuan University, China, in 1998, and the PhD degree in computer science from Huazhong University of Science and Technology, China in 2005. He is now a professor at the College of Computer Science and Electronic Engineering, Hunan University, China. His current research interests include parallel and distributed computing, operating systems, cloud computing and systems security. He has published more than 70 papers in journals and conferences, such as IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Computers, FGCS, ASPLOS, VLDB, PACT, IPDPS, IWQoS, and ICPP. He is a member of the IEEE and the ACM.