# Preprocessing Parallelization for the ALT-Algorithm

Genaro Peque Jr.(✉), Junji Urata, and Takamasa Iryo

Department of Civil Engineering, Kobe University, Kobe, Japan
gpequejr@panda.kobe-u.ac.jp, urata@person.kobe-u.ac.jp, iryo@kobe-u.ac.jp



**Abstract.** In this paper, we improve the preprocessing phase of the ALT algorithm through parallelization. ALT is a preprocessing-based, goal-directed speed-up technique that uses **A*** (A star), **L**andmarks and **T**riangle inequality which allows fast computations of shortest paths (SP) in large-scale networks. Although faster techniques such as arc-flags, SHARC, Contraction Hierarchies and Highway Hierarchies already exist, ALT is usually combined with these faster algorithms to take advantage of its goal-directed search to further reduce the SP search computation time and its search space. However, ALT relies on landmarks and optimally choosing these landmarks is NP-hard, hence, no effective solution exists. Since landmark selection relies on constructive heuristics and the current SP search speed-up is inversely proportional to landmark generation time, we propose a parallelization technique which reduces the landmark generation time significantly while increasing its effectiveness.



**Keywords:** ALT algorithm · Shortest path search
Large-scale traffic simulation


## 1 Introduction

### 1.1 Background

The computation of shortest paths (SP) on graphs is a problem with many real world applications. One prominent example is the computation of the shortest path between a given origin and destination called a single-source, single-target problem. This problem is usually encountered in route planning in traffic simulations and is easily solved in polynomial time using Dijkstra's algorithm [1] (assuming that the graph has non-negative edge weights). Indeed, road networks can easily be represented as graphs and traffic simulations usually use edge (road) distances or travel times as edge weights which are always assumed positive.

In large-scale traffic simulations where edge weights are time-dependent, one is normally interested in computing shortest paths in a matter of a few milliseconds. Specifically, the shortest path search is one of the most computationally intensive parts of a traffic simulation due to the repeated shortest path calculation of each driver departing at a specific time. This is necessary because a driver needs to consider the


© Springer International Publishing AG, part of Springer Nature 2018


Y. Shi et al. (Eds.): ICCS 2018, LNCS 10862, pp. 89–101, 2018.
https://doi.org/10.1007/978-3-319-93713-7_7

effects of the other drivers who departed earlier in his/her route's travel time. However, faster queries are not possible using only Dijkstra or A* (A star) search algorithms. Even the most efficient implementation of Dijkstra or A* isn't sufficient to significantly reduce a simulation's calculation time. Thus, speed-up techniques that preprocess input data [2] have become a necessity. A speed-up technique splits a shortest path search algorithm into two phases, a preprocessing phase and a query phase. A *preprocessing phase* converts useful information on the input data, done before the start of the simulation, to accelerate the *query phase* that computes the actual shortest paths. There are many preprocessing based variants of Dijkstra's algorithm such as the ALT algorithm [3], Arc-Flags [4], Contraction Hierarchies [5], Highway Hierarchies [6] and SHARC [7] among others. These variants are normally a combination of different algorithms that can easily be decomposed into the four basic ingredients that efficient speed-up techniques belong to [8], namely, Dijkstra's algorithm, landmarks [3, 9], arc-flags [10, 11] and contraction [6]. The combination is usually with a goal-directed search algorithm such as the ALT algorithm that provides an estimated distance or travel time from any point in the network to a destination at the cost of additional preprocessing time and space. A few examples are the (i) L-SHARC [8], a combination of landmarks, contraction and arc-flags. L-SHARC increases performance of the approximate arc-flags algorithm by incorporating the goal-directed search, (ii) Core-ALT [12], a combination of contraction and ALT algorithm. Core-ALT reduces the landmarks' space consumption while increasing query speed by limiting landmarks in a specific "core" level of a contracted network, and the (iii) Highway Hierarchies Star (HH*) [13], a combination of Highway Hierarchies (HH) and landmarks which introduces a goal-directed search to HH for faster queries at the cost of additional space.

The ALT algorithm is a goal-directed search proposed by Golberg and Harrelson [3] that uses the A* search algorithm and distance estimates to define node potentials that direct the search towards the target. The A* search algorithm is a generalization of Dijkstra's algorithm that uses node coordinates as an input to a potential function to estimate the distance between any two nodes in the graph. A good potential function can be used to reduce the search space of an SP query, effectively. In the ALT algorithm, a potential function is defined through the use of the triangle inequality on a carefully selected subset of nodes called landmark nodes. The distance estimate between two nodes is calculated using the landmark nodes' precomputed shortest path distances to each node using triangle inequality. The maximum lower bound produced by one of the landmarks is then used for the SP query. Hence, ALT stands for ***A****, ***L***andmarks and ***T***riangle inequality. However, a major challenge for the ALT algorithm is the landmark selection. Many strategies have been proposed such as *random*, *planar*, *Avoid*, *weightedAvoid*, *advancedAvoid*, *maxCover* [3, 9, 13, 14] and as an integer linear program (ILP) [15]. In terms of the landmark generation times of these proposed strategies, *random* is the fastest while ILP is the slowest, respectively. For query times using the landmarks produced by these strategies, *random* is the slowest while ILP is the fastest, respectively. The trend of producing better landmarks at the expense of additional computation times have always been true. Moreover, the aforementioned landmark selection strategies weren't designed to run in parallel. The question is whether a parallel implementation

of a landmark selection algorithm can increase landmark efficiency while only slightly increasing its computation cost.

Our aim is to use the ALT algorithm in the repeated calculation of drivers' shortest paths in a large-scale traffic simulation. The traffic simulator is implemented in parallel using a distributed memory architecture. By reducing the preprocessing phase through parallelization, we are able to take advantage of the parallelized implementation of the traffic simulator and the architecture's distributed memory. Additionally, if we can decrease the landmark generation time, it would be possible to update the landmark lower bounds dynamically while some central processing units (CPUs) are waiting for the other CPUs to finish.

### 1.2 Contribution of This Paper

Since the ALT algorithm is commonly combined with other faster preprocessing techniques for additional speed-up (i.e. goal-directed search), we are motivated in studying and improving its preprocessing phase. Our results show that the parallelization significantly decreased the landmark generation time and SP query times.

### 1.3 Outline of This Paper

The paper is structured as follows. In the following section, the required notation is introduced. Additionally, three shortest path search algorithms, namely, Dijkstra, A* search and the ALT algorithm, are presented. Section 3 is dedicated to the description of the landmark preprocessing techniques, our proposed landmark generation algorithm and its parallel implementation. In Sect. 4, the computational results are shown where our conclusions, presented in Sect. 5, are drawn from.

## 2 Preliminaries

A graph is an ordered pair $G = (V, E)$ which consists of a set of vertices, $V$, and a set of edges, $E \subset V \times V$. Sometimes, $(u, v) \in E$, will be written as $e \in E$ to represent a link. Additionally, vertices and edges will be used interchangeably with the terms nodes and links, respectively. Links can either be composed of an unordered or ordered pairs. When a graph is composed of the former, it is called an undirected graph. If it composed of the latter, it is called a directed graph. Throughout this paper, only directed graphs are studied. For a directed graph, an edge $e = (u, v)$ leaves node $u$ and enters node $v$. Node $u$ is called the tail while the node $v$ is called the head of the link and its weight is given by the cost function $c\colon E \to \mathbb{R}^+$. The number of nodes, $|V|$, and the number of links, $|E|$, are denoted as $n$ and $m$, respectively.

A path from node $s$ to node $t$ is a sequence, $\left(v_0, v_1, \ldots, v_{k-1}, v_k\right)$, of nodes such that $s = v_0, t = v_k$ and there exists an edge $\left(v_{i-1}, v_i\right) \in E$ for every $i \in \{1, \ldots, k\}$. A path from $s$ to $t$ is called simple if no nodes are repeated on the path. A path's cost is defined as,

$$dist(s,t) := \sum_{i=1}^{k} c_{(v_{i-1},v_i)}. \tag{1}$$

A path of minimum cost between nodes $s$ and $t$ is called the $(s,t)-$ shortest path with its cost denoted by $dist^*(s,t)$. If no path exists between nodes $s$ and $t$ in $G$, $dist(s,t) := \infty$. In general, $c(u,v) \neq c(v,u), \forall(u,v) \in E$ so that $dist(s,t) \neq dist(t,s)$. One of the most well-known algorithms used to find the path and distance between two given nodes is the Dijkstra's algorithm.

### 2.1 Dijkstra's Algorithm

In Dijkstra's algorithm, given a graph with non-negative edge weights, a source (origin) and a target (destination), the shortest path search is conducted in such a way that a "Dijkstra ball" slowly grows around the source until the target node is found.

More specifically, Dijkstra's algorithm is as follows: during initialization, all node costs (denoted by $g$) from the source, $s$, to all the other nodes are set to infinity (i.e. $g(w) = \infty, \forall w \in V \backslash \{s\}$). Note that $g(s) = 0$. These costs are used as *keys* and are inserted into a minimum-based priority queue, $PQ$, which decides the order of each node to be processed. Take $w = argmin_{u \in PQ} g(u)$ from the priority queue. For each node $v \in PQ$ subject to $(w,v) \in E$, if $g(v) > g(w) + c(w,v)$, set $g(v) = g(w) + c(w,v)$. Repeat this process until either the target node is found or $PQ$ is empty. This means that the algorithm checks all adjacent nodes of each processed node, starting from the source node, which is the reason for the circular search space or "Dijkstra ball".

Although Dijkstra's algorithm can calculate the shortest path from $s$ to $t$, its search speed can still be improved by using other network input data such as node coordinates. This is the technique used by the A* algorithm described in the next subsection.

### 2.2 A* (A Star) Search Algorithm

The A* search algorithm is a generalization of Dijkstra's algorithm. A* search algorithm uses a potential function, $\pi_t : V \to \mathbb{R}^+$, which is an estimated distance from an arbitrary node to a target node.

Consider the shortest path problem from a source node to a target node in the graph and suppose that there is a potential function, $\pi_t$, such that $\pi_t(u)$ provides an estimate of the cost from node $u$ to a given target node, $t$. Given a function $g : V \to \mathbb{R}^+$ and a priority function, $PQ$, defined by $PQ(u) = g(u) + \pi_t(u)$, let $g^*(u)$ and $\pi_t^*(u)$ represent the length of the $(s,u)-$ shortest path and $(u,t)-$ shortest path, respectively. Note that $g^*(u) = dist^*(s,u)$ and $\pi_t^*(u) = dist^*(u,t)$, so that $PQ^*(u) = g^*(u) + \pi_t^*(u) = dist^*(s,t)$. This means that the next node that will be taken out of $PQ$ belongs to the $(s,t)-$ shortest path. If this holds true for the succeeding nodes until the target node is reached, its search space starting at node $u$ will only consist of the shortest path nodes. However, if $PQ(u) = g^*(u) + \pi_t(u) = dist(s,t)$ for some $u$, then its search space will increase depending on how bad the estimates for each node is, $\pi_t(u)$.

In the A* search algorithm, the value used for the potential function is the Euclidean or Manhattan distance based on the nodes' coordinates. Given $\pi_t$, the reduced link cost

is defined as $c_{e,\pi_t} = c(u, v) - \pi_t(u) + \pi_t(v)$. We say that $\pi_t$ is feasible if $c_{e,\pi_t} \geq 0$ for all $e \in E$. The feasibility of $\pi_t$ is necessary for the algorithm to produce a correct solution. A potential function, $\pi$, is called valid for a given network if the A* search algorithm outputs an $(s, t)-$ shortest path for any pair of nodes.

The potential function can take other values coming from the network input data. The algorithm described in the next subsection takes preprocessed shortest path distances to speed-up the SP search.

### 2.3 ALT Algorithm

The ALT algorithm is a variant of the A* search algorithm where landmarks and the triangle inequality are used to compute for a feasible potential function. Given a graph, the algorithm first preprocesses a set of landmarks, $L \subset V$ and precomputes distances from and to these landmarks for every node $w \in V$. Let $L = \left\{l_1, \ldots, l_{|L|}\right\}$, based on the triangle inequality, $|x + y| \leq |x| + |y|$, two inequalities can be derived, $dist^*(u, v) \geq dist^*(l, v) - dist^*(l, u)$ (see Fig. 1) and $dist^*(u, v) \geq dist^*(u, l) - dist^*(v, l)$ for any $u, v, l \in V$. Therefore, potential functions denoted as,

$$\pi_t^+(v) = dist^*(v, l) - dist^*(t, l), \tag{2}$$

$$\pi_t^-(v) = dist^*(l, t) - dist^*(l, v), \tag{3}$$

can be defined as feasible potential functions.

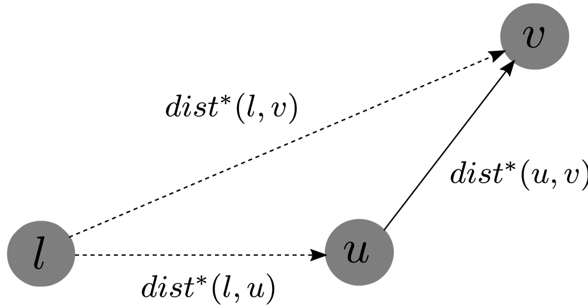


**Fig. 1.** A triangle inequality formed by a landmark $l$ and two arbitrary nodes $u$ and $v$

To get good lower bounds for each node, the ALT algorithm can use the maximum potential function from the set of landmarks, i.e.,

$$\pi_t(v) = \max_{l \in L} \left\{\pi_t^+(v), \pi_t^-(v)\right\} \tag{4}$$

A major advantage of the ALT algorithm over the A* search algorithm in a traffic simulation is its potential function's input flexibility. While the A* search algorithm can only use node coordinates to estimate distances, the ALT algorithm accepts either travel time or travel distance as a potential function input which makes it more robust with

respect to different metrics [12]. This is significant because traffic simulations usually use travel times as edge weights where a case of a short link length with a very high travel time and a long link length with a shorter travel time can occur. Moreover, the ALT algorithm has been successfully applied to social networks [16, 17] where most hierarchal, road network oriented methods would fail.

One disadvantage of the ALT algorithm is the additional computation time and space required for the landmark generation and storage, respectively.

# 3 Preprocessing Landmarks

Landmark selection is an important part of the ALT algorithm since good landmarks produce good distance estimates to the target. As a result, many landmark selection strategies have been developed to produce good landmark sets. However, as the landmark selection strategies improved, the preprocessing time also increased. Furthermore, this increase in preprocessing time exceeds the preprocessing times of faster shortest path search algorithms to which the ALT algorithm is usually combined with [13]. Thus, decreasing the landmark selection strategy's preprocessing time is important.

## 3.1 Landmark Selection Strategies

Finding a set, $k$, of good landmarks is critical for the overall performance of the shortest path search. The simplest and most naïve algorithm would be to select a landmark, uniformly at random, from the set of nodes in the graph. However, one can do better by using some criteria for landmark selection. An example would be to randomly select a vertex, $\hat{v}_1$, and find a vertex, $v_1$, that is farthest away from it. Repeat this process $k$ times where for each new randomly selected vertex, $\hat{v}_i$, the algorithm would select a the vertex, $v_i$, farthest away from all the previously selected vertices, $\{v_1, \dots, v_{i-1}\}$. This landmark generation technique is called *farthest* proposed by Goldberg and Harrelson [3].

In this paper, a landmark selection strategy called *Avoid*, proposed by Goldberg and Werneck [9], is improved and its preprocessing phase is parallelized. In the *Avoid* method, a shortest path tree, $T_r$, rooted at node $r$, selected uniformly at random from the set of nodes, is computed. Then, for each node, $v \in V$, the difference between $dist^*(r, v)$ and its lower bound for a given $L$ is computed (e.g. $dist^*(r, v) - dist^*(l, v) + dist^*(l, r)$). This is the node's weight which is a measure of how bad the current cost estimates are. Then, for each node $v$, the size is calculated. The size, $size(v)$, depends on $T_v$, a subtree of $T_r$ rooted at $v$. If $T_v$ contains a landmark, $size(v)$ is set to 0, otherwise, the $size(v)$ is calculated as the sum of the weights of all the nodes in $T_v$. Let $w$ be the node of maximum size, traverse $T_w$ starting from $w$ and always follow the child node with the largest size until a leaf node is reached. Make this leaf a new landmark (see Fig. 2). This process is repeated until a set with $k$ landmarks is generated.

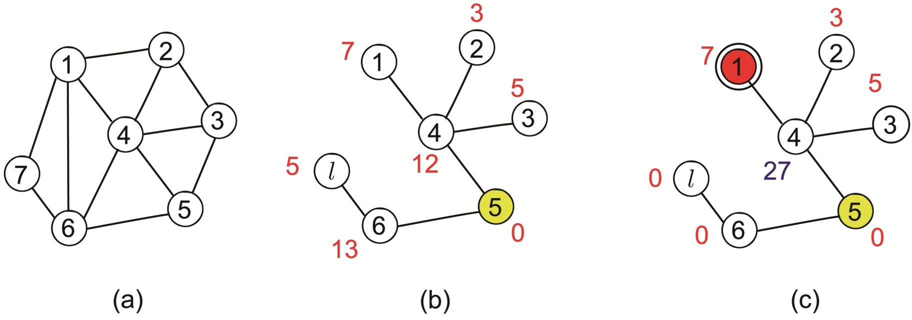


**Fig. 2.** The *Avoid* method. (a) A sample network with 7 nodes is shown. In (b), node 5 is randomly selected and a shortest path tree, $T_5$, is computed. Then, the distance estimate is subtracted by the shortest path cost for each node. These are the nodes' weights (shown in red). In (c), node sizes are computed. Since node 7 is a landmark and the subtree, $T_6$, has a landmark, both sizes are set to 0. Starting from the node with the maximum size (node 4), traverse the tree deterministically until a leaf is reached (node 1) and make this a landmark (Color figure online)

A variant of *Avoid* called the *advancedAvoid* was proposed to try to compensate for the main disadvantage of *Avoid* by probabilistically exchanging the initial landmarks with newly generated landmarks using *Avoid* [13]. It had the advantage of producing better landmarks at the expense of an additional computation cost. Another variant called *maxCover* uses *Avoid* to generate a set of $4k$ landmarks and uses a scoring criterion to select the best $k$ landmarks in the set through a local search for $\lfloor \log_2 k + 1 \rfloor$ iterations. This produced the best landmarks at the expense of an additional computation cost greater than the additional computation cost incurred by *advancedAvoid*.

We improve *Avoid* by changing the criteria of the shortest path tree to traverse deterministically. Rather than just consider the tree with the maximum size, $T_w$, the criteria is changed to select the tree with the largest value when size is multiplied by the number of nodes in the tree, i.e. $size(v) \times |T_v|$, where $|T_v|$ denotes the number of nodes in the tree $T_v$ [15]. This method prioritizes landmarks that can cover a larger region without sacrificing much quality. Additionally, following the *advancedAvoid* algorithm, after $k$ landmarks are generated, a subset $\hat{k} \subset k$ is removed uniformly at random and the algorithm then continues to select landmarks until $k$ landmarks are found.

To increase landmark selection efficiency without drastically increasing the computation time, the algorithm, which we call *parallelAvoid*, is implemented in parallel using the C++ Message Passing Interface (MPI) standard. In *parallelAvoid*, each CPU, $p_j$, generates a set of $k$ landmarks using the method outlined in the previous paragraph. These landmark sets are then evaluated using a scoring criterion that determines the effectiveness of the landmark set. The score determined by each CPU is sent to a randomly selected CPU, $p_k$, which then determines the CPU with the maximum score, $\hat{p}_j$. The CPU $p_k$ sends a message informing $\hat{p}_j$ to broadcast its landmark set to all the other CPUs including $p_k$ (see Fig. 3).

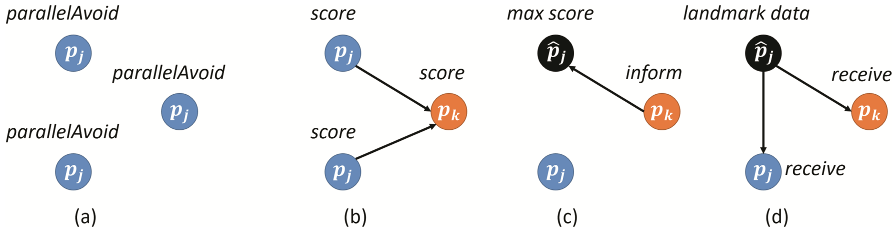


**Fig. 3.** The *parallelAvoid* landmark strategy algorithm. (a) The CPUs, $p_j$, generates landmarks using the *parallelAvoid* algorithm. (b) The score for each landmark set is sent to CPU $p_k$ where the maximum score is determined. (c) The CPU $p_k$ then informs the CPU with the maximum score, $\hat{p}_j$, to send its landmark data to all the other CPUs. (d) The CPU, $\hat{p}_j$, sends its landmark data to all the other CPUs

Hence, in terms of the query phase calculation time, the hierarchy of the landmark generation algorithms introduced above are as follows,

$$Avoid > advancedAvoid > parallelAvoid \geq maxCover > ILP, \tag{5}$$

There can be a case where *parallelAvoid* produces better landmarks, thus better query times, than *maxCover*. This case happens when the CPUs used to generate landmarks significantly exceed the $\lfloor \log_2 k + 1 \rfloor$ iterations used by the *maxCover* algorithm.

For the landmark preprocessing phase calculation time, the hierarchy of the same landmark generation algorithms are as follows,

$$Avoid < advancedAvoid \leq parallelAvoid < maxCover < ILP. \tag{6}$$

### 3.2 Landmark Set Scoring Function

In order to measure the quality of the landmarks generated by the different landmark generation algorithms, a modified version of the maximum coverage problem [14] is solved and its result is used as the criterion to score the effectiveness of a landmark set.

The maximum coverage problem is defined as follows: Given a set of elements, $\{a_1, \ldots, a_r\}$, a collection, $S = \{S_1, \ldots, S_p\}$, of sets where $S_i \subset \{a_1, \ldots, a_r\}$ and an integer $k$, the problem is to find a subset $S^* \subset S$ with $|S^*| \leq k$ so that a maximum number of elements $a_i$ covered is maximal, i.e., max $\bigcap_{S_i \in S^*} S_i$. Such a set $S^*$ is called a set of maximum coverage.

In order to find the maximum coverage, a query of selected source and target pairs are carried out using Dijkstra's algorithm. A $|V|^3 \times |V|$ matrix with one column for each node in each search space of the Dijkstra query and one row for each node, $v \in V$, interpreted as a potential landmark, $l_i$, is initialized with zero entries. The ALT algorithm is carried out on the same source and target pairs queried by the Dijkstra's algorithm. Then for each node in the Dijkstra algorithm's search space, if the ALT algorithm doesn't visit the node, $v$, using $l_i$ (i.e. $l_i$ would exclude $v$ from the search space), the entry for the column assigned to the node $v$ is set to 1 (see Fig. 4). Selecting $k$ rows so that a maximum number of columns are covered is equivalent to the maximum coverage problem.

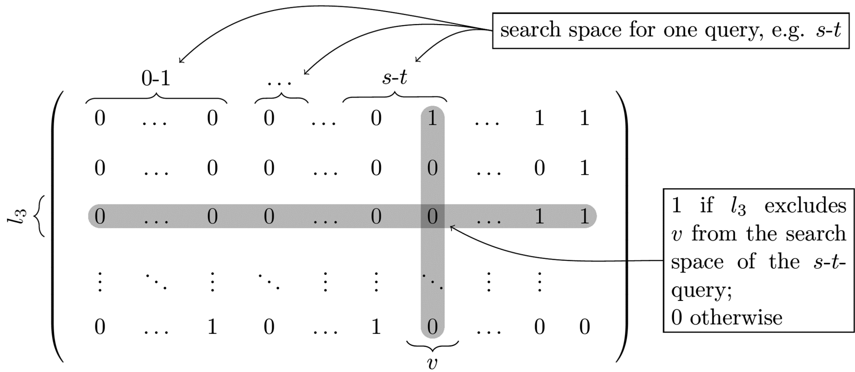


**Fig. 4.** The matrix for the landmark selection problem interpreted as a maximum coverage problem [14]

For our case, the maximum coverage problem is modified to only consider a subset of the origin-destination pairs used in the traffic simulation for the $(s, t)-$ query. Additionally, each row is composed of the $k$ landmarks found by *parallelAvoid* rather than all the nodes in the node set. The modified maximum coverage problem is then defined as the landmark set that has the maximum number of columns in the matrix that is set to 1. Furthermore, the number of columns set to 1 are summed up and then used as the score of the landmark set. In this modified maximum coverage problem, the landmark set with the highest score is used for the shortest path search.

## 4 Computational Results

The network used for the computational experiments is the Tokyo Metropolitan network which is composed of 196,269 nodes, 439,979 links and 2210 origin-destination pairs. A series of calculations were carried out and averaged over 10 executions for the *Avoid*, *advanceAvoid*, *maxCover* and *parallelAvoid* algorithms using Fujitsu's K computer. A landmark set composed of $k = 4$ landmarks with $\hat{k} = 2$ and $\hat{k} = 0$ for the *advanceAvoid* and *parallelAvoid*, respectively, were generated for each landmark selection strategy.

The K computer is a massively parallel CPU-based supercomputer system at the Advanced Institute of Computational Science, RIKEN. It is based on a distributed memory architecture with over 80,000 compute nodes where each compute node has 8 cores (SPARC64™ VIIIfx) and 16 GB of memory. Its node network topology is a 6D mesh torus network called the tofu (**to**rus **fu**sion) interconnect.

The performance measures used are the execution times of the algorithms and the respective query times of its SP searches using the landmark sets that each of the algorithms have generated. For *parallelAvoid*, its parallel implementation was also measured using different number of CPUs for scalability.

### 4.1 Execution and Query Times

The average execution times of the *Avoid* (A), *advanceAvoid* (AA), *maxCover* (MC) and *parallelAvoid* (PA) algorithms generating 4 landmarks and the average query times of each SP search using the generated landmarks are presented in the Table 1 below.

**Table 1.** Averaged execution times of the landmark selection strategies and averaged query times of each SP searches in seconds.

| Number of CPUs | A | AA | MC | PA | Query time |
|---|---|---|---|---|---|
| 1 (A) | 124.270 | —— | —— | —— | 0.1382 |
| 1 (AA) | —— | 180.446 | —— | —— | 0.1253 |
| 1 (MC) | —— | —— | 521.602 | —— | 0.1081 |
| 2 (PA) | —— | —— | —— | 133.829 | 0.1390 |
| 4 (PA) | —— | —— | —— | 134.515 | 0.1282 |
| 8 (PA) | —— | —— | —— | 140.112 | 0.1128 |
| 16 (PA) | —— | —— | —— | 143.626 | 0.1102 |
| 32 (PA) | —— | —— | —— | 157.766 | 0.1078 |
| 64 (PA) | —— | —— | —— | 160.941 | 0.1044 |
| 128 (PA) | —— | —— | —— | 163.493 | 0.1022 |
| 256 (PA) | —— | —— | —— | 169.606 | 0.1004 |
| 512 (PA) | —— | —— | —— | 178.278 | 0.0998 |
| 1024 (PA) | —— | —— | —— | 180.661 | 0.0986 |

The table above shows that algorithms took at least 30 s to select a landmark. The *advanceAvoid* algorithm took a longer time as it selected 6 landmarks (i.e. $k = 4$ and $\hat{k} = 2$). The *maxCover* algorithm took the longest time as it generated 16 landmarks and selected the best 4 landmarks through a local search. The local search was executed $\lfloor \log_2 k + 1 \rfloor$ which is equal to 2 in this case. The *parallelAvoid* algorithm's execution (landmark generation) time slowly increased because of the increasing number of CPUs that the CPU, $\hat{p}_j$, had to share landmark data with.

In terms of query times, the algorithms follow Eq. (5) up to 16 CPUs. At 32 CPUs, *parallelAvoid*'s query performance is better than *maxCover*'s query performance. Note that 32 CPUs mean that 32 different landmark sets were generated by *parallelAvoid* which is twice the number of landmarks generated by the *maxCover* algorithm. Moreover, the table also shows that the number of CPUs is directly proportional to the landmark generation time and inversely proportional to the query time. The former is due to the overhead caused by communication which is expected. As the number of communicating CPUs increase, the execution time of the algorithm also increases. While for the latter, as the number of CPUs increase the possibility of finding a better solution also increases which produces faster query times.

### 4.2 Algorithm Scalability

A common task in high performance computing (HPC) is measuring scalability of an application. This measurement indicates the application's efficiency when using an increasing number of parallel CPUs.

The *parallelAvoid* algorithm belongs to the weak scaling case where the problem size assigned to each CPU remains constant (i.e. all CPUs will generate a set of landmarks which consumes a lot of memory). This is very efficient when used with the K computer's distributed memory architecture. In the Fig. 5 above, a major source of overhead is data transfer. The data transfer overhead is caused by the transfer of landmark data from CPU, $\hat{p}_j$, to all the other CPUs. This was implemented using the MPI_Bcast command which has a tree-based structure that has a logarithmic complexity. Hence, it can be noticed that by using a logarithmic scale on the x-axis, the weak scaling is significantly affected by the data transfer overhead.

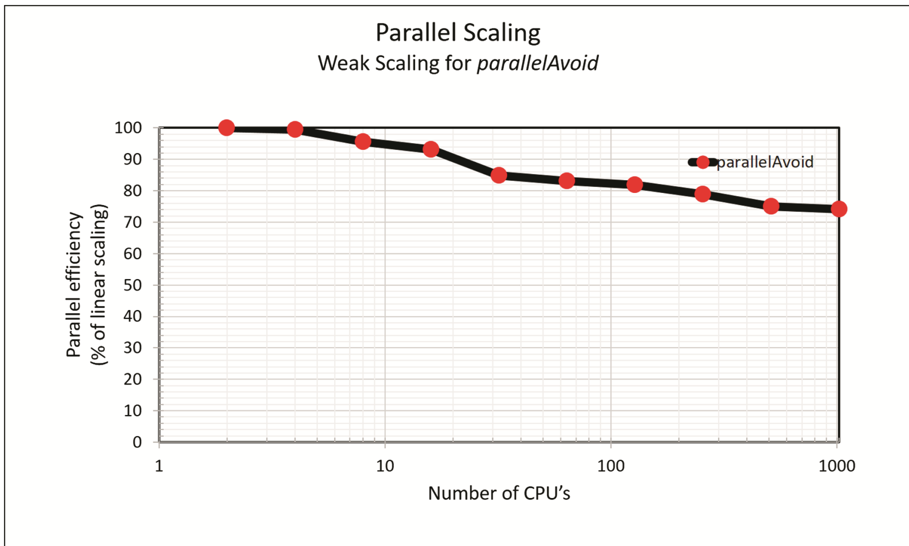


**Fig. 5.** Weak scaling for the *parallelAvoid* algorithm.

## 5 Conclusions

In this paper, we have presented a parallelized ALT preprocessing algorithm called the *parallelAvoid*.

We have shown that this algorithm can increase the landmark's efficiency while significantly accelerating the preprocessing time using multiple CPUs. By using many CPUs, it is possible to obtain a better landmark set in a significantly lesser amount of time which produces faster query times. Compared to the *maxCover* algorithm, it is limited only to the number of CPUs rather than the number of local search iterations. This is important since the ALT algorithm is usually combined with other preprocessing

algorithms to take advantage of its goal-directed search. Moreover, the parallelization technique doesn't sacrifice landmark quality in exchange for preprocessing speed unlike the ALP (A*, landmarks and polygon inequality) algorithm [18, 19], a generalization of the ALT algorithm or the *partition-corners* method used in [20, 21].

Additionally, results show that the major cause of overhead is the landmark data transfer. This is because the data transfer of the landmark data from the CPU with the highest score to the other CPUs use a tree-like structure which has a logarithmic complexity.

**Acknowledgement.** This work was supported by Post K computer project (Priority Issue 3: Development of Integrated Simulation Systems for Hazard and Disaster Induced by Earthquake and Tsunami).

This research used computational resources of the K computer provided by the RIKEN Advanced Institute for Computational Science through the HPCI System Research Project (Project ID: hp170271).

## References

1. Dijkstra, E.: A note on two problems in connexion with graphs. Numer. Math. **1**, 269–271 (1959)
2. Wagner, D., Willhalm, T.: Speed-up techniques for shortest-path computations. In: Thomas, W., Weil, P. (eds.) STACS 2007. LNCS, vol. 4393, pp. 23–36. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-70918-3_3
3. Goldberg, A., Harrelson, C.: Computing the shortest path: A* search meets graph theory. Technical report (2004)
4. Gutman, R.: Reach-based routing: a new approach to shortest path algorithms optimized for road networks. In: Proceedings 6th Workshop on Algorithm Engineering and Experiments (ALENEX), pp. 100–111. SIAM (2004)
5. Geisberger, R., Sanders, P., Schultes, D., Delling, D.: Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In: WEA, pp. 319–333 (2008)
6. Sanders, P., Schultes, D.: Highway hierarchies hasten exact shortest path queries. In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 568–579. Springer, Heidelberg (2005). https://doi.org/10.1007/11561071_51
7. Bauer, R., Delling, D.: SHARC: fast and robust unidirectional routing. ACM J. Exp. Algorithmics **14** (2009)
8. Delling, D., Wagner, D.: Pareto paths with SHARC. In: Vahrenhold, J. (ed.) SEA 2009. LNCS, vol. 5526, pp. 125–136. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02011-7_13
9. Goldberg, A., Werneck, R.: Computing point-to-point shortest paths from external memory. In: Proceedings Workshop on Algorithm Engineering and Experiments (ALENEX 2005). SIAM (2005)
10. Köhler, E., Möhring, R.H., Schilling, H.: Acceleration of shortest path and constrained shortest path computation. In: Nikoletseas, S.E. (ed.) WEA 2005. LNCS, vol. 3503, pp. 126–138. Springer, Heidelberg (2005). https://doi.org/10.1007/11427186_13
11. Lauther, U.: An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. In: Geoinformation und Mobilitat - von der Forschung zur praktischen Anwendung, vol. 22, pp. 219–230. IfGI prints (2004)

12. Bauer, R., Delling, D., Sanders, P., Schieferdecker, D., Schultes, D., Wagner, D.: Combining hierarchical and goal-directed speed-up techniques for Dijkstra's algorithm. In: McGeoch, C.C. (ed.) WEA 2008. LNCS, vol. 5038, pp. 303–318. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68552-4_23
13. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Highway hierarchies star. In: 9th DIMACS Implementation Challenge (2006)
14. Peque Jr., G., Urata, J., Iryo, T.: Implementing an ALT algorithm for large-scale time-dependent networks. In: Proceedings of the 22nd International Conference of Hong Kong Society for Transport Studies, 9–11 December 2017
15. Fuchs, F.: On Preprocessing the ALT Algorithm. Master's thesis, University of the State of Baden-Wuerttemberg and National Laboratory of the Helmholtz Association, Institute for Theoretical Informatics (2010)
16. Mehlhorn, K., Sanders, P.: Algorithms and Data Structures: The Basic Toolbox. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-77978-0
17. Tretyakov, K., Armas-Cervantes, A., García-Bañuelos, L., Vilo, J., Dumas, M.: Fast fully dynamic landmark-based estimation of shortest path distances in very large graphs. In: Proceedings 20th CIKM Conference, pp. 1785–1794 (2011)
18. Campbell Jr., N.: Computing shortest paths using A*, landmarks, and polygon inequalities. arXiv:1603.01607 [cs.DS] (2016)
19. Campbell Jr., N.: Using quadrilaterals to compute the shortest path. arXiv:1603.00963 [cs.DS] (2016)
20. Efentakis, A., Pfoser, D.: Optimizing landmark-based routing and preprocessing. In: Proceedings of the Sixth ACM SIGSPATIAL International Workshop on Computational Transportation Science (IWCTS 2013), New York, pp. 25–30, November 2013
21. Efentakis, A., Pfoser, D., Vassiliou, Y.: SALT. A unified framework for all shortest-path query variants on road networks. arXiv:1411.0257 [cs.DS] (2014)