Proceedings of the Twenty-Sixth International Conference on
Automated Planning and Scheduling (ICAPS 2016)

Evaluation of Auction-Based Multi-Robot Routing by Parallel Simulation

Akihiro Kishimoto
IBM Research, Ireland
akihirok@ie.ibm.com

Kiyohito Nagano
Department of Complex and Intelligent Systems
Future University Hakodate, Japan
k_nagano@fun.ac.jp

Abstract

Auction methods are a promising approximation approach for distributed routing including multi-robot routing, where targets on a map need to be allocated to agents while a team objective is satisfied. While many algorithms based on sequential single-item (SSI) auctions have been presented, they are currently evaluated by serial simulation where agents serially calculate their bids on a single machine.

We consider a scenario where a bidding algorithm incurs significant computational overhead due to on-demand calculations of the shortest distances on a road map. We evaluate the bidding algorithm under parallel simulations where agents perform bid calculations simultaneously on a parallel machine, and reveal that the algorithm suffers from severe synchronization overhead ignored by serial simulation. We also present the broadcasting and speculation techniques to alleviate such synchronization overhead.

Our empirical results on multi-robot routing variants show that both techniques improve the efficiency of parallelization, and speculation achieves more significant improvement.

Introduction

Multi-agent coordination has been a challenging planning problem which has many applications. Examples of applications include multi-agent routing on a map, such as search and rescue operations and robot routing.

The multi-robot routing (MRR) problem (Lagoudakis et al. 2005) has been a typical testbed for multi-agent coordination. In MRR, agents and targets are allocated initially on a map. Then, the targets need to be assigned to the agents while an objective as a team of agents is satisfied, such as minimizing the response time of the slowest agent and minimizing the total resource usage among the agents.

MRR is a difficult combinatorial optimization problem when the agent and target sizes are large. It is therefore often infeasible to optimally solve MRR.

Approximation algorithms are used to heuristically return non-optimal but valid solutions in feasible time. Auction algorithms especially based on sequential single-item (SSI) auctions have been extensively studied as a promising approximation approach to solve MRR, by regarding agents and targets as bidders and auction items, respectively.

Copyright © 2016, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

When agents compute bids, they typically use the shortest distances among the locations of the agents and targets. When the map size is small, the shortest distance values of all location pairs on the map are easily precomputed and stored in memory. The existing work aiming to improve the solution quality assumes the availability of precomputed tables, e.g., (Koenig et al. 2007; Zheng, Koenig, and Tovey 2006). The shortest distance of a location pair is retrieved by a table lookup that only needs the $O(1)$ time complexity.

When the map size is large, the shortest distances need to be calculated on demand, since the precomputed table does not fit into memory of each agent. Other examples of the necessity of the on-demand distance calculation include the scenario where the map changes, e.g., many roads are blocked off due to unexpected disasters such as floods and earthquakes. Replacing the shortest distances with heuristic ones computed by the Manhattan-distance-based heuristic reduces the memory requirement, but returns solutions with poor qualities (Kishimoto and Sturtevant 2008).

The shortest distances are computed by running *pathfinding search* on the map typically represented as a graph. A node and an edge in the graph respectively correspond to a location and a small segment of a road in the map. Pathfinding employs a search-based approach such as A* (Hart, Nilsson, and Raphael 1968) and Dijkstra's algorithm (Dijkstra 1959), and examines the graph with the non-trivial size. Pathfinding search is invoked many times when agents calculate bids. Kishimoto and Sturtevant (2008) show that the shortest-distance calculation is computationally intensive on a large map, even with the basic SSI auction algorithm that considers only a small number of locations pairs.

SSI auctions have been considered to allow for both efficient communication and parallelism, as agents can compute their bids in parallel (Koenig, Keskinocak, and Tovey 2010). However, the existing work evaluates the algorithms only under serial simulations where only one agent runs serially at a time on a single machine. The actual parallel efficiency of the auction methods has not been well studied yet.

We elucidate the scaling behavior of a bidding algorithm based on SSI auctions with pathfinding search under parallel simulations, where agents calculate bids simultaneously on a parallel machine. In our MRR using a road map, agents suffer from severe computational overhead caused by on-demand pathfinding search. We reveal that the bidding algo-

rithm scales very poorly under parallel simulations, because of the idle times of the agents at the synchronization point in each round. This overhead has been hidden under serial simulations performed in the previous work.

We additionally present two methods to reduce synchronization overhead: One is to broadcast the current best bid cost when it is updated. The other is to perform speculation after agents finish their bidding. Our empirical results on MRR demonstrate that both broadcasting and speculation successfully scale up the parallel performance, and speculation is more crucial to improve the efficiency.

Multi-Robot Routing Problem

We define a multi-robot routing problem (MRR) that slightly extends that of Lagoudakis et al. (2005).

MRR consists of M homogeneous agents and N targets placed on a road map. Each target has a source location and a destination, and needs to be collected at its source first and then delivered to its destination by one agent.

Notations A and T denote the sets of agents and targets, respectively. Let AS, TS and TD be the sets of start locations of the agents, sources and destinations of the targets, respectively. The cost of moving from location l to location m, called the *edge cost* or the *cost of the edge*, is denoted by $c(l,m)$ $(l,m \in AS \cup TS \cup TD)$. However, like (Kishimoto and Sturtevant 2008), $c(l,m)$ *is not precomputed*: $c(l,m)$ must be computed by performing pathfinding search in the road map represented as a *road map graph*.[1] The latitude and longitude based locations in the road map are transformed into the nodes of the road map graph. In addition, roads are divided into many segments annotated with their transition costs. When there is a segment of a road between two locations in the road map, the road map graph contains an edge which connects the two nodes corresponding to these locations of the road map.

Pathfinding needs to run state-space search in the road map graph such A* (Hart, Nilsson, and Raphael 1968) and Dijkstra's algorithm (Dijkstra 1959), and incurs non-negligible computational overhead. In the worst case, in order to calculate $c(l,m)$ of one location pair, pathfinding may need to examine $|RN|$ nodes and $|RE|$ edges, where RN and RE are the sets of nodes and edges in the road map graph, respectively. For example, the Open Street Map[2] of Dublin in Ireland contains roughly 300,000 nodes and 320,000 edges.

Each agent $a_i \in A$ collects and delivers all targets allocated to a_i by constructing a path from a_i's start location to the destination of the final target to deliver. The path cost of a_i is defined as the total edge cost on that path.

We deal with the following typical team objectives:

- MINIMAX: Minimize the maximum path cost of any single agent.
- MINISUM: Minimize the path cost sum over all agents.

[1] Note that the node set RN of the road map graph is different from $S := AS \cup TS \cup TD$. S is a very small subset of RN.

[2] https://www.openstreetmap.org/

Algorithm 1 The SSI auction algorithm

```
Require: Agent size M and a set of targets T
1: N = |T|
2: for (i = 1; i ≤ N; i = i + 1) do
3:     bbc = ∞;
4:     for (j = 1; j ≤ M; j = j + 1) do
5:         // Calculate a bid cost bc and target t for agent a_j
6:         (bc,t) = CalcBidandTarget(j, T)
7:         if (bc < bbc) then
8:             bbc = bc; bt = t; w = j
9:     T = T\{bt}
```

We consider two scenarios about the capacity of the agent. *Without capacity constraints*, each agent accepts an unlimited number of targets. On the other hand, *with capacity constraints*, each agent can carry at most k targets at a time where k is a constant. If a_i uses up its capacity k, a_i needs to drop off one of the targets at its destination before accepting a new target. We empirically evaluate the performance in both scenarios. However, when describing algorithms, we assume no capacity constraints, since these algorithms are easily adapted to the scenario with capacity constraints.

Related Work

We review the literature on SSI auctions applied to MRR. See (Koenig et al. 2006; Koenig, Keskinocak, and Tovey 2010) for a more comprehensive survey.

Lagoudakis et al. (2005) prove that MINISUM and MINIMAX are NP-hard. To find good solutions that are close to optimal ones in a short time, auction methods have been studied especially in the form of SSI auctions, e.g., (Tovey et al. 2005; Lagoudakis et al. 2004).

There are approaches to improve the solution quality of SSI auctions, e.g., (Zheng, Koenig, and Tovey 2006; Koenig et al. 2007; 2008). However, we review only the basic bidding algorithm, because these improved versions use precomputed tables for $c(l,m)$ and have not been combined with on-demand pathfinding yet. In addition, even with the basic bidding algorithm, our implementation as well as (Kishimoto and Sturtevant 2008) already incur severe overhead caused by pathfinding. The improved bidding algorithms usually need an extra set of location pairs the basic bidding algorithm does not consider (e.g., changing the order of several locations in the current path). They would suffer from more excessive overhead, thus opening up a new research opportunity that is beyond the scope of the paper.

Let M and N be the number of agents and targets, respectively. SSI auctions consist of N rounds of auctions where agents are the bidders and targets are the items to bid on. In each round, SSI auctions auction off only one target currently unallocated to any agent: the agent bidding the lowest value on a target wins that target with an arbitrary tie-breaking rule. Due to this winner determination rule, in each round, each agent bids the lowest bid on a single target, and sends the bid to the server. After the server receives the bids from all agents, it broadcasts the winner in that round.

Algorithm 1 shows the pseudo-code of the SSI auction algorithm performed by serial simulation. T preserves a set

of unallocated targets in the current round. In *CalBidandTarget*, agent a_j calculates a bid cost bc on one unallocated target in T.[3] The winner in each round is stored in w. In parallel simulations, the code to exchange messages between the agents and the server is needed, such as notifying the winner w and the target bt to the agents from the server.

Lagoudakis et al. (2005) present bidding rules with theoretical analysis for a few objectives including MINISUM and MINIMAX. The PATH rule is a common baseline when new algorithms are developed, e.g., (Koenig et al. 2008; Zheng, Koenig, and Tovey 2006). Let T_i be the set of targets agent a_i has already obtained, and U be the set of unallocated targets. Under the PATH rule, a_i directly constructs the shortest path cost $SPC(a_i, T_i)$ for agent a_i to visit all sources and destinations of targets in T_i, starting from a_i's start location. Because computing $SPC(a_i, T_i)$ itself is NP-hard, a heuristic method, typically the insertion heuristic for TSP (Lawler et al. 1985), is used to approximate SPC. The insertion heuristic greedily inserts only one location in the best place of the the path currently constructed. Notation SPC is still used to denote such heuristic SPC in the paper. Agent a_i's bid cost is defined as:

$$\min_{t \in U} SPC(a_i, T_i \cup \{t\}) \quad \text{(MINIMAX)}$$

$$\min_{t \in U} SPC(a_i, T_i \cup \{t\}) - SPC(a_i, T_i) \quad \text{(MINISUM)}$$

Note that each agent bids only for the target t that minimizes MINIMAX/MINISUM.

The main focus in the above is to improve the solution quality with precomputed $c(l, m)$. FASTBID (Kishimoto and Sturtevant 2008) is the only attempt to combine the bidding algorithm with pathfinding. FASTBID reveals that the computational overhead of on-demand shortest-distance calculation can be large, as the map size as well as the agent and target size grow. In order to alleviate the overhead for pathfinding, they also present an approach which hierarchically abstracts the grid map graph (game map with many pixels) based on (Sturtevant and Buro 2005), and performs pathfinding on such an abstract graph. On the other hand, we test the implementation on the road map which has different characteristics than the grid map. It is still unknown whether Sturtevant and Buro's (2005) approach is effective on road maps or not. Additionally, the bidding algorithms combined with on-demand pathfinding in the road map would have real-world applications such as search and rescue operations, and taxi sharing (Ma, Zheng, and Wolfson 2013; Nakashima et al. 2014) in the long run.

All the work surveyed in this section evaluates the runtimes of the bidding algorithms under serial simulations without considering the synchronization overhead caused by the agents. This is the overhead we attempt to alleviate.

Implementation Design for SSI Auctions under Serial Simulations

We discuss our serial implementation of an SSI auction algorithm combined with on-demand pathfinding. Our implementation is based on FASTBID (Kishimoto and Sturtevant 2008), which is currently the only available algorithm in our MRR setting where $c(l, m)$ is not precomputed.

We incorporate FASTBID's techniques that bound edge cost computations based on lowerbounds of the edge costs as well as the second-best bid information sent by the server. Therefore, our implementation design is not particularly new. However, while FASTBID uses the TREE rule which builds a minimum spanning tree to calculate the bids, our implementation is based on the PATH rule which directly generates a path. In practice, there is a consensus that PATH tends to return better solutions than TREE (Lagoudakis et al. 2005), and most followup research about improving the solution quality (with the precomputed tables of $c(l, m)$) is based on PATH (Koenig, Keskinocak, and Tovey 2010). In addition, we employ a different pathfinding algorithm than FASTBID, because we use the road map which has different characteristics from the grid map for which FASTBID is designed.

Bounding Edge Cost Computations

Let $path(a_i)$ be agent a_i's current path generated by the sources and destinations of the targets obtained by a_i, and $p = (j, k, t)$ be a tuple called the *insertion index tuple* (IIT) which indicates that the source and destination of an unallocated target t are inserted to the j-th and k-th indexes of $path(a_i)$ by the TSP insertion heuristic. In the beginning of the round, for each unallocated target, a_i enumerates all possible IITs in $path(a_i)$. The insertion heuristic considers the capacity constraints and the fact that a target must be collected before a_i drops it off. Let P be the set of all IITs. We consider only the paths generated by P to calculate SPC.

Let $pc(p)$ be the path cost generated by $path(a_i)$ and IIT p, and $lb(p)$ be a lowerbound of $pc(p)$. We run pathfinding search to compute the edge costs that are necessary to calculate $pc(p)$.

Assume that pathfinding search attempts to calculate an edge cost from location l to m. Pathfinding search examines the road map graph, and gradually improves the lowerbound and upperbound of the edge cost as the search progresses. When the lowerbound of the edge cost equals the upperbound, pathfinding search returns the exact value of the edge cost (see the next subsection).

Because pathfinding search improves a lowerbound of the edge cost, we do not always need compute the exact value of $pc(p)$: We use threshold θ to stop pathfinding if $lb(p) > \theta$ holds. With θ, we aim to compute the exact value of $pc(p_{best})$ where $p_{best} = \text{argmin}_{p \in P} pc(p)$, and prove $pc(p_{best}) \leq lb(q)$ for any $q \in P \backslash \{p_{best}\}$. We reduce the search effort of pathfinding by increasing the number of cases where it has only to compute $lb(q)$.

Specifically, as in FASTBID, a priority queue Q sorts all $p \in P$ in increasing order of $lb(p)$. Let p_1 and p_2 be the best and second best IITs in Q. Then, p_1 is dequeued from Q at each step, and $lb(p_1)$ is improved by performing pathfinding with $\theta = lb(p_2)$. If the exact value of $pc(p_1)$ is calculated, p_1 is proven to be the best, and calculating $pc(p_{best})$ terminates. Otherwise, p_1 is enqueued back in Q and the above step is repeated.

[3] If an agent loses an auction in the current round, and the best strategy for that agent in the next round is to bid the same bid cost on the same target, that agent does not perform any bid calculation.

Pathfinding based on Contraction Hierarchy

We incorporate Contraction Hierarchy (CH) of Geisberger et al. (2008). CH has been an essential building block for optimal pathfinding on a road map in practice. Unlike FASTBID, CH's pathfinding performs bidirectional Dijkstra search in the road map graph to compute $c(l,m)$ from location l to location m. Forward and backward searches alternately examine the state spaces defined by different sets of states based on CH.

Let l and m be the source and destination in the road map graph, respectively. Forward search preserves the *g-value* of node n defined as the sum of the transition costs from l to reach n in the road map graph. If more than one path leads to n, the smallest g-value is preserved. Forward search uses an *open list* which orders nodes in increasing order of the g-value, and a *closed list* which preserves the nodes that have been already examined. Forward search first places l in the open list. Then, it repeats the procedure of removing one node n that has the smallest g-value in the open list, storing n in the closed list, generating n's successor nodes, and inserting these successor nodes to the open list.

Backward search preserves the *g-value* of node n defined as the sum of the transition costs from n to m, and performs the procedure that is similar to forward search except that backward search starts with m in a direction toward l.

When pathfinding has not established the exact value of $c(l,m)$ yet, a lowerbound of $c(l,m)$ is calculated by taking the minimum g-value in the open lists of forward and backward searches. When forward and backward searches meet at node s, the path from l to m via s establishes an upperbound of $c(l,m)$. When the lowerbound of $c(l,m)$ equals the upperbound of $c(l,m)$, the exact value of $c(l,m)$ is successfully computed.

At agent's start location, we run only forward search. For the source and destination of each target, we run both forward and backward searches. Both searches are performed incrementally and limited with the threshold. They preserve their closed and open lists when they stop searching. When additional pathfinding is necessary, search is resumed with the closed and open lists preserved before.

Bounding Edge Cost Computation Further with the Second Best Bid

Let t be a target auctioned off in the round, and sbb be the best bid of all except ones on t. As Kishimoto and Sturtevant (2008) describe, our server implementation broadcasts sbb as well as the winner and the obtained target. Such a "second best" bid further bounds edge cost computations in the next round: Agents must send the bid costs smaller than sbb to win. By performing the bid cost computation bounded with sbb, each agent either bids the exact bid cost or gives up obtaining a target in each round. Assume that agents have exact bid costs without performing any pathfinding. Even if they have no chance to win with these bid costs, they still send such exact bid costs to the server. This is because these bid costs may contribute to bounding sbb more tightly in subsequent rounds.

The above approach is more clearly explained with an example which consists of four agents (a_1, a_2, a_3, a_4) and three targets (t_1, t_2, t_3). In the first round, if a_1, a_2, a_3, and a_4 bid 3, 4, 5, and 7 on t_1, t_1, t_2 and t_3, respectively, a_1 wins t_1. Then, in the second round, a_3 bids 5 on t_2 again, since aiming to obtain t_2 still remains best for a_3 according to the PATH rule. Therefore, in the second round, the bid costs of a_1 and a_2 need to be smaller than 5 to win. If a_1's pathfinding search proves that its bid cost is at least 5, a_1 gives up obtaining any target. Agent a_2 performs an analogous procedure. Although a_4 has no chance of winning the target, it still bids 7 on target t_3. This is important because a_4's bid cost of 7 contributes to bounding the bid cost computations of the other agents to 7 in the third round, for example, if a_1 bids 4 on t_2 and a_2 gives up in the second round.

Improving SSI Auctions under Parallel Simulations

Under parallel simulations, the server waits until receiving the bid costs from all agents. That is, there is a synchronization point where the agents need to synchronize in each round. This can be a cause of increasing idle times of the agents, especially when the runtimes to calculate bids are unbalanced among the agents. For example, the bid cost of an agent may remain unchanged between the previous and next rounds (i.e. that agent needs no calculation), while the agent that has won the target needs to compute a new bid. We introduce two techniques, *broadcasting* and *speculation*, to alleviate this issue.

Broadcasting

The server preserves the current best bid cost *bbc* during each round. An agent that is still calculating its bid cost *bc* has no chance of obtaining a target if $bbc \leq bc$ holds. Broadcasting broadcasts *bbc* to all agents when the server improves *bbc*.[4] The agent that is currently computing its *bc* periodically checks if *bbc* arrives by using a non-blocking message-receiving procedure. If the agent receives *bbc*, it updates its bounding condition for edge cost computations to be able to give up bidding in case that $bbc \leq bc$ is proven. This way, broadcasting achieves more effective pruning for the agents that are still working on bid cost calculation, which results in reducing the idle times of the agents that have already submitted their bids.

Broadcasting incurs additional communication overhead. As a trade-off, we limit the maximum number of broadcasts to a constant B in each round. In this way, the total number of messages the server and agents send is at most $BMN + MN + MN = (B+2)MN$, where M is the number of agents and N is the number of targets. In contrast, this number is $2MN$ for the original SSI auctions.

Algorithm 2 shows the pseudo-code of the server. Lines 7–9 are added to support broadcasting. *RecvBidCostFromAgent* receives a bid cost bc and a target t from agent a. *BcastToAllAgents* sends out the current best bid cost *bbc* for round i to all agents. *CalcSecondBestBid* returns the second

[4]This could be implemented in such a way that agents directly communicate with one another about the updated *bbc*. However, for the sake of simplicity, we introduce communication via the server.

Algorithm 2 Server algorithm with broadcasting

```
Require: Broadcasting size B and target size N
 1: for (i = 1; i ≤ N; i = i + 1) do
 2:     bbc = ∞; count = 0;
 3:     repeat
 4:         (bc, t, a) = RecvBidCostFromAgent()
 5:         if (bc ≠ "give up" ∧ bc < bbc) then
 6:             bbc = bc; bt = t; w = a
 7:             count = count + 1
 8:             if count ≤ B then
 9:                 BcastToAllAgents(bbc,i)
10:     until (all agents finish bidding for round i)
11:     sbb = CalcSecondBestBid()
12:     BcastBidResult(w, bt, sbb)
```

best bid cost sbb as explained in the previous section. *Bcast-BidResult* broadcasts the winner w, the target bt obtained by w, and the second best bid sbb.

Speculation

Assume that agent a_k waits for the winner information for round i. In speculation, rather than sitting idle, a_k speculates to calculate the bid cost for round $i+1$ as if the server had already determined the winner for round i. The server algorithm remains the same, therefore, the amount of communication is unchanged.

We present two speculation strategies. *Optimistic speculation* assumes that a_k successfully obtains target t to bid on in round i. Agent a_k updates its path by including t's source and destination, then speculation is performed. In contrast, *pessimistic speculation* assumes that t is allocated to another agent. Therefore, a_k speculates to find another unallocated target except t by using the current path.

Assume that a_k finishes speculatively calculating the bid cost and target (bc, t) for round $i+1$. In case that a_k's assumption to the winner in round i is correct, a_k has only to submit (bc, t) immediately when round $i+1$ starts. Otherwise, it recalculates the bid in round $i+1$ with the correct winner and target information sent by the server. Even in this case, a_k successfully reduces the runtime of the overall bid calculation in the remaining rounds by using the exact edge costs and improved lowerbounds of the edge costs calculated by speculation.

Speculation increases the frequency of completing the SPC calculation immediately after all IITs are inserted in the priority queue. In case that a_k knows the exact bid cost, it sends not the give-up message but the exact bid cost, even if a_k has no chance to win. As we describe in the previous section, this is because a_k's bid cost may bound the bid cost computations of the other agents. Hence, speculation contributes to obtaining better SPC calculation bounds than straightforward parallelization, which is an additional factor of achieving speedups.

If a_k completes its speculation for round $i+1$ but has not received the winner yet, it speculates the bid for round $i+2$. Thus, a_k continues speculation for the future rounds until the winner for round i becomes available.

Algorithms 3 shows the pseudo-code of the agent algorithm that incorporates speculation. For the sake of simplicity, the code of broadcasting is omitted here.

Algorithm 3 Agent algorithm with speculation but without broadcasting

```
Require: Target set T and agent a_k
 1: θ = ∞; N = |T|
 2: for (i = 1; i ≤ N; i = i + 1) do
 3:     if (IsSpeculatedBidAvailable(a_k,i)) then
 4:         (bc,t) = GetBidandTarget(a_k, i)
 5:     else
 6:         (bc,t) = CalcBidandTarget(a_k, T,θ)
 7:     SendBidandTarget(a_k,i,bc,t)
 8:     U = T
 9:     for (j = i + 1; j ≤ N; j = j + 1) do
10:         if (IsSpeculationOptimistic()) then
11:             UpdateAgentPath(a_k,t)
12:         U = U\{t}
13:         (bc,t) = CalcBidandTarget(a_k,U,∞)
14:         if (IsBidResultAvailable(i)) then
15:             RevertAgentPath(a_k,i)
16:             exit loop
17:     (w,bt,sbb) = RecvBidResult(i)
18:     if (a_k = w) then
19:         UpdateAgentPath(w,bt)
20:     T = T\{bt}
21:     θ = sbb
```

If the bid cost bc and the target to bid on t is already available in round i, which is checked by *IsSpeculatedBidAvailable*, a_k just retrieves such bc and t with *GetBidandTarget* and sends them to the server with *SendBidandTarget*. Otherwise, a_k calculates SPC and selects a target to bid on from the set of unallocated targets T. *CalcBidandTarget* does this task. Note that θ is the bounding condition based on the second best bid.

Optimistic/pessimistic speculation is performed in lines 9–16. Let U be the set of unallocated targets used for speculation. In case of optimistic speculation, a_k updates its path with t's source and destination. *UpdateAgentPath* performs this step. When speculation is performed, the algorithm does not bound the bid calculation with the second best bid (see the value of ∞ passed to CalcBidandTarget in line 13). Speculation is repeated until the winner information is arrived at a_k from the server. *IsBidResultAvailable* checks such a message and *RecvBidResult* receives the winner w, the target bt obtained by w, and the second best bid sbb. *RevertAgentPath* reverts a_k's path back to the path for round i.

Note that *CalcBidandTarget* needs to periodically check if the server determines the winner for round i, indicating that IsBidResultAvailable is called in inside this function.

Experimental Results

The bidding algorithms we have discussed and applied to MRR were implemented in C++ with the MPI library (Snir and Gropp 1998) on top of Open Source Routing Machine[5], which includes a freely available, efficient CH implementation. All the experiments were performed on a PC cluster

[5] http://project-osrm.org/

Table 1: Speedups of each method for MINIMAX (150 targets)

(a) Capacity = ∞

Method	Number of agents and speedups 10	20	30	40
BASELINE	3.32	3.94	5.83	6.34
PESSI	**7.87**	9.10	13.66	17.58
BCAST(1)	4.20	5.35	8.39	11.07
BCAST(5)	4.16	5.78	9.50	12.20
BCAST(1)+ PESSI	7.83	9.22	13.97	**18.81**
BCAST(5)+ PESSI	7.83	**9.27**	**14.05**	18.39
BCAST(5)+ OPTI	**7.87**	9.22	14.01	17.20

(b) Capacity=4

Method	Number of agents and speedups 10	20	30	40
BASELINE	3.26	4.03	5.88	7.60
PESSI	8.20	9.50	13.53	17.89
BCAST(1)	4.10	5.31	8.34	11.37
BCAST(5)	4.02	5.79	9.30	12.69
BCAST(1)+ PESSI	8.22	9.59	14.12	18.72
BCAST(5)+ PESSI	8.24	9.61	**14.22**	**19.06**
BCAST(5)+ OPTI	**8.25**	**9.63**	14.20	18.04

Table 2: Speedups of each method for MINISUM (150 targets)

(a) Capacity = ∞

Method	Number of agents and speedups 10	20	30	40
BASELINE	1.99	2.33	3.17	3.84
PESSI	**2.74**	3.70	5.40	7.02
BCAST(1)	2.39	2.79	3.82	4.69
BCAST(5)	2.41	2.93	4.06	4.88
BCAST(1)+ PESSI	2.73	3.73	**5.70**	7.41
BCAST(5)+ PESSI	2.73	3.74	**5.70**	**7.62**
BCAST(5)+ OPTI	**2.74**	**3.88**	5.63	7.27

(b) Capacity=4

Method	Number of agents and speedups 10	20	30	40
BASELINE	2.47	2.48	3.17	3.89
PESSI	7.08	7.66	10.12	14.17
BCAST(1)	3.27	3.07	3.84	4.75
BCAST(5)	3.32	3.21	4.03	4.91
BCAST(1)+ PESSI	**7.22**	**8.03**	10.64	14.29
BCAST(5)+ PESSI	7.12	7.96	**11.20**	14.58
BCAST(5)+ OPTI	7.20	8.00	10.87	**15.06**

whose compute node consists of 12 CPU cores (Intel Xeon X5690 at 3.47GHz) and 80GB of RAM. We used up to four compute nodes to perform serial and parallel simulations. In performing serial simulations, we followed the same evaluation methodology as (Kishimoto and Sturtevant 2008): The simulations were performed serially on a single CPU core without sharing any edge cost computation among the agents. In contrast, when performing parallel simulations, we allocated one CPU core to the server and one core to each agent. Except the fact that these agents run in parallel in a completely decentralized way, we used the same evaluation methodology as serial simulations.

While the number of targets was always set to 150, the number of agents was varied from 10 to 40. These numbers were chosen to generate non-trivial problems. In addition, we also prepared the problems without and with capacity constraints. With capacity constraints, each agent was allowed to carry at most 4 targets at a time. In each category, we prepared 20 problems by randomly allocating agents and targets on the map of Dublin in Ireland.

We evaluate the efficiency of parallel algorithms by measuring their speedups defined as:

$$\text{speedup} = \frac{\text{runtime by a serial algorithm}}{\text{runtime by a parallel algorithm}}.$$

We report an average speedup for each parallel implementation. We prepared the following versions:

- BASELINE: No enhancement is added.
- BCAST(1): Broadcasting is added. The number of broadcasting messages B is set to one per round.
- BCAST(5): Broadcasting is added. B is set to five.
- PESSI: Pessimistic speculation is added.
- OPTI: Optimistic speculation is added.

A combination of the above methods is denoted by "+".

Tables 1 and 2 show speedup values of parallel algorithms. Our results about BASELINE clearly demonstrate that the advantage of distributed computation among agents does not allow for efficient parallelism in our MRR setting. For example, in solving MINIMAX without capacity constraints, BASELINE obtains the speedup values of 3.32, 3.94, 5.83, and 6.34 for 10, 20, 30, and 40 agents, respectively. For MINISUM without capacity constraints, BASELINE has very limited scalability: its speedup values are 1.99, 2.33, 3.17, and 3.84 for 10, 20, 30, and 40 agents, respectively. For MINISUM, a few agents tend to obtain most of the targets. These agents need to perform more intensive edge cost computations, due to many places to insert the sources and destinations of the unallocated targets in the current path (i.e., pathfinding needs to be performed more frequency). Let bt be the target agent a_k bids on. The phenomenon that a few agents take most of the targets is caused by the fact that the bid cost is based on the proximity of bt to a_k's current path for MINISUM. Obtaining additional targets increases a chance of having unallocated targets located nearby a_k's path. Once a_k generates a long path, a_k keeps winning the targets, thus becoming *overworked*. Although introducing capacity constraints alleviates the issue, BASELINE still suffers from performance degradation caused by such unbalanced work.

Both broadcasting and pessimistic speculation outperform BASELINE. With 40 agents, PESSI yields 1.83–3.64 times larger speedups than BASELINE, while this number is 1.22–1.92 for broadcasting. Combining speculation with

Table 3: Average runtimes of serial simulations (150 targets, seconds)

(a) Capacity = ∞

	Number of agents and runtimes			
	10	20	30	40
MINIMAX	153.43	285.50	424.81	558.63
MINISUM	157.11	250.52	344.63	437.17

(b) Capacity=4

	Number of agents and runtimes			
	10	20	30	40
MINIMAX	163.85	304.00	432.87	565.58
MINISUM	154.99	278.57	399.91	513.93

broadcasting yields the largest speedups with 40 agents but only performs slightly better than speculation itself (1-9% reduction in runtimes). Speculation increases the chance of agents submitting their bids immediately after the rounds start, and tends to reduce room for the agents to perform pruning with broadcasting.

BCAST(5) performs slightly better than BCAST(1). A larger performance difference is observed with a larger number of agents for MINIMAX. This indicates that the bid costs the server receives second or later are often smaller than the earliest one if many agents participate in bidding. The performance difference is smaller for MINISUM, since the overworked agents tend to become winners and since they send their bids later than the others.

When agents keep speculating for many future rounds, PESSI and OPTI may have a very different set of edges to consider. However, we observe that their speedups are similar. We hypothesize that this is partly because the performance is currently limited by the amount of work performed by the overworked agents, especially for MINISUM.

Table 3 shows average runtimes of serial simulations, measured in seconds. Our implementation needs 150–560 seconds to solve one problem consisting of 10–40 agents and 150 targets. The insertion heuristic needs almost 0 second if $c(l, m)$ is precomputed. These runtimes, therefore, indicate high computational overhead of on-demand pathfinding search even with the computationally inexpensive bidding algorithm based on the insertion heuristic. Combining on-demand pathfinding with other bidding algorithms would be challenging future work, since these algorithms would require additional edge cost computations that are not performed by the insertion-heuristic-based bidding algorithm. For example, the 2-opt and 3-opt improvements (Lawler et al. 1985; Tovey et al. 2005) modify the order of locations in the current path and introduce new edges of which costs need to be computed by pathfinding.

Figures 1 and 2 show graphs of work and idle times for each agent in solving a typical problem. The vertical axis is the agent ID (1–40) and the horizontal axis is the time. The horizontal red line indicates that an agent is calculating a bid. Otherwise, the agent sits idle at the synchronization points. Table 4 shows the ratio of the maximum and average idle times to the runtime. These results clearly indicate that BASELINE suffers from large synchronization overhead caused by the idle times until the server determines the winner in each round. For example, on average, agents sit idle for 70% and 83% of their runtimes for MINIMAX and MINISUM, respectively. The synchronization overhead is more clearly observed for MINISUM. Additionally, agents tend to become more idle in later rounds of the bidding procedure.

Table 4: Ratio of maximum and average idle times of agents to the runtime (40 agents, 150 targets, capacity = ∞)

	MINIMAX		MINISUM	
Method	Max	Ave	Max	Ave
BASELINE	0.80	0.70	0.89	0.83
BCAST(5)	0.66	0.51	0.86	0.79
BCAST(5)+ PESSI	0.25	0.03	0.55	0.31

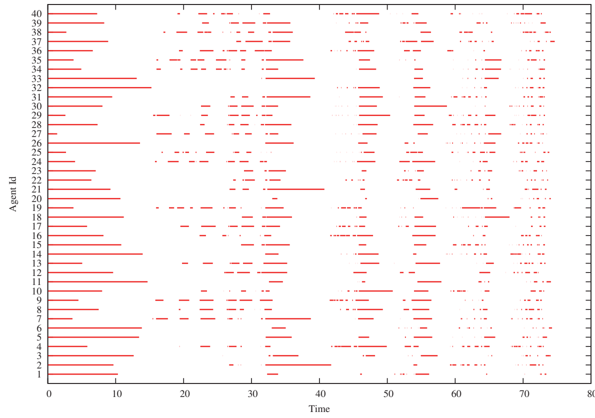
BCAST(5) reduces the synchronization overhead by making agents give up bidding earlier when their attempts become useless. However, they still have large average idle time ratios of 51% and 79% for MINIMAX and MINISUM.

Further introducing PESSI significantly reduces the synchronization overhead, especially for MINIMAX, resulting in yielding larger speedups (18.4 and 19.1 fold average speedups with 40 agents without and with capacity constraints, respectively). On the other hand, PESSI still incurs idle times caused by the case where an agent finishes its speculation until the final round, therefore, has no work to perform. This scenario becomes evident for MINISUM (see BCAST(5)+ PESSI in Figure 2). After 25 seconds, there are only a few agents constantly working on bid cost calculations, while the other agents remain almost always idle. This is closely related to the fact that a few agents tend to obtain most of the targets. We leave resolving this issue as an important extension as future work.

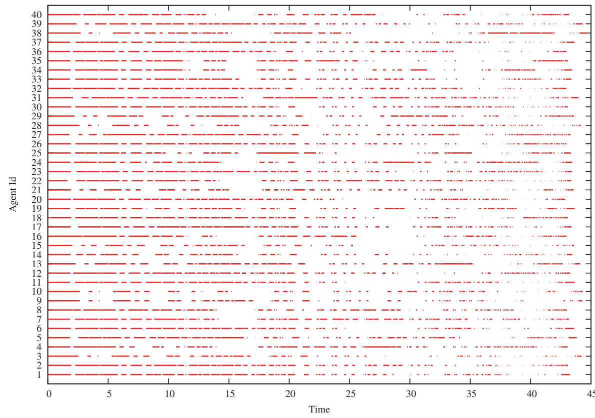
Conclusions and Future Work

We evaluated the performance of bidding algorithms combined with pathfinding under parallel simulations, and revealed that the synchronization overhead hidden in the evaluation of previous work is a cause of limited scalability. We, therefore, attempted to alleviate the synchronization overhead by performing speculation and broadcasting. Our results on MRR demonstrated that both speculation and broadcasting improve the efficiency of agents' bid calculations, and speculation impacts more on the parallel performance.

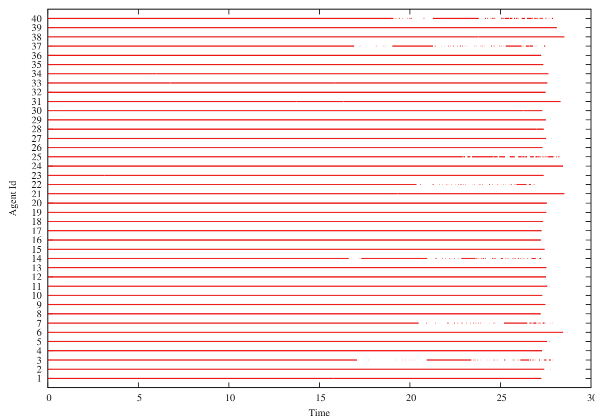
There are numerous extensions to this work, including: (1) developing algorithms to further improve the parallel efficiency, especially for MINISUM, (2) combining other bidding algorithms with pathfinding efficiently and evaluating them under parallel simulations, (3) investigating ideas for the scenario where more complicated conditions are involved, such as more constraints including time windows and dynamic changes to the target set, and (4) considering a scenario where communication failures and delays occur between the server and the robots.



(a) BASELINE

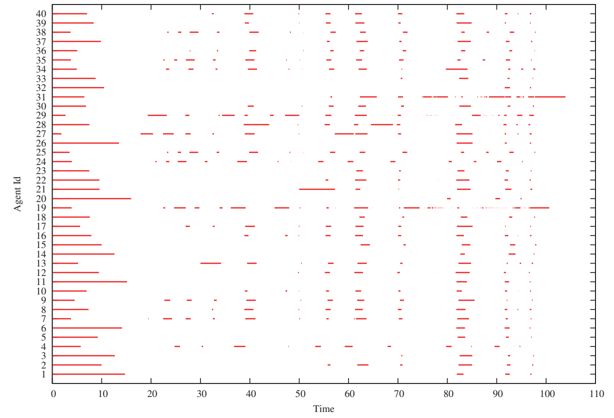


(b) BCAST(5)

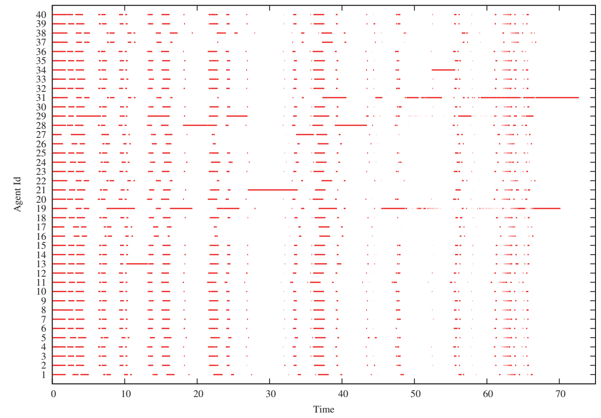


(c) BCAST(5)+ PESSI

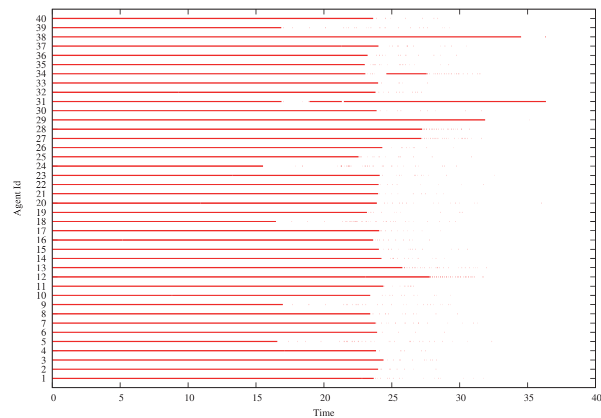
Figure 1: Work and idle times of agents for MINIMAX (40 agents, 150 targets, capacity = ∞)



(a) BASELINE



(b) BCAST(5)



(c) BCAST(5)+ PESSI

Figure 2: Work and idle times of agents for MINISUM (40 agents, 150 targets, capacity = ∞)

References

Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik* 1:269–271.

Geisberger, R.; Sanders, P.; Schultes, D.; and Delling, D. 2008. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Proceedings of the 7th International Conference on Experimental Algorithms (WEA'08)*, 319–333.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 4(2):100–107.

Kishimoto, A., and Sturtevant, N. 2008. Optimized algorithms for multi-agent routing. In *AAMAS*, 1585–1588.

Koenig, S.; Tovey, C. A.; Lagoudakis, M. G.; Markakis, V.; Kempe, D.; Keskinocak, P.; Kleywegt, A. J.; Meyerson, A.; and Jain, S. 2006. The power of sequential single-item auctions for agent coordination. In *AAAI*, 1625–1529.

Koenig, S.; Tovey, C.; Zheng, X.; and Sungur, I. 2007. Sequential bundle-bid single-sale auction algorithms for decentralized control. In *IJCAI*, 1359–1365.

Koenig, S.; Zheng, X.; Tovey, C.; Borie, R.; Kilby, P.; Markakis, V.; and Keskinocak, P. 2008. Agent coordination with regret clearing. In *AAAI*, 101–107.

Koenig, S.; Keskinocak, P.; and Tovey, C. 2010. Progress on agent coordination with cooperative auctions. In *AAAI*, 1713–1717.

Lagoudakis, M. G.; Berhault, M.; Koenig, S.; Keskinocak., P.; and Kleywegt, A. J. 2004. Simple auctions with performance guarantees for multi-robot task allocation. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS'04)*, volume 1, 1957–1962.

Lagoudakis, M.; Markakis, V.; Kempe, D.; Keskinocak, P.; Koenig, S.; Kleywegt, A.; Tovey, C.; Meyerson, A.; and Jain, S. 2005. Auction-based multi-robot routing. In *Proceedings of the International Conference on Robotics: Science and Systems*, 343–350.

Lawler, E. L.; Lenstra, J. K.; Kan, A. H. G. R.; and Shmoys, D. B. 1985. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. Wiley Series in Discrete Mathematics and Optimization.

Ma, S.; Zheng, Y.; and Wolfson, O. 2013. T-share: A large-scale dynamic taxi ridesharing service. In *Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE'13)*, 410–421.

Nakashima, H.; Sano, S.; Hirata, K.; Shiraishi, Y.; Matsubara, H.; Kanamori, R.; Koshiba, H.; and Noda, I. 2014. One cycle of smart access vehicle service development. In *Proceedings of the 2nd International Conference on Serviceology*, 152–157.

Snir, M., and Gropp, W. 1998. *MPI: The Complete Reference*. MIT Press.

Sturtevant, N., and Buro, M. 2005. Partial pathfinding using map abstraction and refinement. In *AAAI*, 1392–1397.

Tovey, C.; Lagoudakis, M.; Jain, S.; and Koenig, S. 2005. The generation of bidding rules for auction-based robot coordination. In *Multi-Robot Systems: From Swarms to Intelligent Automata*, volume 3, 3–14. Springer.

Zheng, X.; Koenig, S.; and Tovey, C. 2006. Improving sequential single-item auctions. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS'06)*, 2238– 2244.