Decentralized Task Allocation for Dynamic Environments

by

Luke B. Johnson

Submitted to the Department of Aeronautics and Astronautics
in partial fulfillment of the requirements for the degree of

Master of Science in Aeronautics and Astronautics

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Feb 2012

Author .....
Department of Aeronautics and Astronautics
Feb 2, 2012

Certified by .....
Jonathan P. How
Richard C. Maclaurin Professor of Aeronautics and Astronautics
Thesis Supervisor

Accepted by .....
Professor Eytan H. Modiano
Professor of Aeronautics and Astronautics
Chair, Graduate Program Committee

Decentralized Task Allocation for Dynamic Environments

by

Luke B. Johnson

Submitted to the Department of Aeronautics and Astronautics
on Feb 2, 2012, in partial fulfillment of the
requirements for the degree of
Master of Science in Aeronautics and Astronautics

Abstract

This thesis presents an overview of the design process for creating greedy decentralized task allocation algorithms and outlines the main decisions that progressed the algorithm through three different forms. The first form was called the Sequential Greedy Algorithm (SGA). This algorithm, although fast, relied on a large number of iterations to converge, which slowed convergence in decentralized environments. The second form was called the Consensus Based Bundle Algorithm (CBBA). CBBA required significantly fewer iterations than SGA but it is noted that both still rely on global synchronization mechanisms. These synchronization mechanisms end up being difficult to enforce in decentralized environments. The main result of this thesis is the creation of the Asynchronous Consensus Based Bundle Algorithm (ACBBA). ACBBA broke the global synchronous assumptions of CBBA and SGA to allow each agent more autonomy and thus provided more robustness to the task allocation solutions in these decentralized environments.

Thesis Supervisor: Jonathan P. How
Title: Richard C. Maclaurin Professor of Aeronautics and Astronautics

Acknowledgments

I would like to thank my advisor, Prof. Jonathan How, for his infinite patience and guidance throughout the past two and a half years. I would also like to thank Prof. Han-Lim Choi for his invaluable technical advice and guidance throughout my masters work. To my colleagues at the Aerospace Controls Lab, especially Sameera Ponda, Andrew Kopein, Andrew Whitten, Dan Levine, Buddy Michini, Josh Redding, Mark Cutler, Kemal Ure, Alborz Geramifard, Tuna Toksoz and Trevor Campbell, you have been irreplaceable as inspiration for my research and as friends throughout my graduate work.

Finally I would like to thank my friends and family for all of their support in the few waking hours a day I was able spend away from MIT, especially to Aaron Blankstein who joined me on excellent adventures throughout the Boston area, and to my parents who have always encouraged me to do what I love.

This work was sponsored (in part) by the AFOSR and USAF under grant FA9550-08-1-0086 and by Aurora Flight Sciences under SBIR - FA8750-10-C-0107. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.

Contents

1 Background Information on Decentralized Planners **13**
1.1 Introduction 13
1.2 Literature Review 25
1.3 Thesis Contributions 27

2 Task Assignment Problem **29**
2.1 General Problem Statement 29
2.2 Algorithmic Design Goals 34

3 Planning Approaches **37**
3.1 Coupled Element 37
3.2 The Algorithmic Decisions 40
3.3 Sequential Greedy 46
3.4 Consensus-Based Bundle Algorithm (CBBA) 52
3.5 CBBA Analysis 60

4 The Asynchronous Consensus Based Bundle Algorithm **67**
4.1 Asynchronous CBBA (ACBBA) 68
4.2 Asynchronous Replan 83
4.3 The future of ACBBA 85

A Improving the convergence speed of the CBBA algorithm. **87**
A.1 Implementation Details 88
A.2 Results and what they mean for performance 88

References **98**

List of Figures

3-1 Three difficulties with decentralized synchronization. The width of the boxes (labeled as agents 1-3) represents the computation time for the bundle building phase. The black bar above these agents represents the globally set heartbeat time. 61

4-1 Agent decision architecture: Each agent runs two main threads, Listener and BundleBuilder; these two threads interface via the Incoming Buffers – there are two buffers to avoid unexpected message overriding; outgoing buffers manage rebroadcasting of information, triggered by the completion of the BundleBuilder thread. 69

4-2 Message profile for a convergence iteration where all the tasks are released at the start of the program 80

4-3 Message profile for a convergence sequence when 10 tasks seed the environment and pop-up tasks are released every 0.2s after 81

4-4 Message profile for a convergence sequence when 10 tasks seed the environment and pop-up tasks are released every 0.05s after 82

4-5 Monte Carlo simulation results showing comparisons between number of messages, and convergence times 84

A-1 This Plot shows the number of total conflicts as a function of the percentage of the fleet planned for. 89

A-2 This Plot shows the number of total conflicts as a function of the percentage of the fleet planned for. 90

A-3 This Plot shows the number of final conflicts as a function of the percentage of the fleet planned for. 91

List of Tables

3.1 CBBA Action rules for agent i based on communication with agent k regarding task j 57

4.1 ACBBA decision rules for agent i based on communication with agent k regarding task j (z_{uj}: winning agent for task j from agent u's perspective; y_{uj}: winning bid on task j from agent u's perspective; t_{uj}: timestamp of the message agent u received associated with the current z_{uj} and y_{uj}) 75

Chapter 1

Background Information on Decentralized Planners

1.1 Introduction

The multi-agent planning problem in its most basic form is the process of making decisions. Humans solve planning problems continuously, having to make decisions about how and when their day should progress. At some point you have to decide where you are going to send your body (can be thought of as an agent), and what you are going to do when you get there (often can be quantized into tasks.) This process happens continually and is driven by the maximization of some cost function, and, if you are very lucky, you are maximizing some measure of happiness for yourself. Unfortunately, the decisions we make in our daily lives are not constraint free, otherwise, I would choose to be an alien that gains superpowers from exposure to a yellow sun. We have physical constraints (I can't be in 2 places at once), resource constraints (I have to eat every so often to continue function), capability constraints (no matter how hard I try I can't solve abstract algebra problems), and various other constraints (I need money to do almost everything). This process of making decisions about how to plan for agents is the basic process that researchers and engineers in task planning are working to automate. The solutions to these planning problems are useful in many aspects of our lives: from how airliners are routed at airports,

to the order in which processes are being scheduled for execution on my cpu as this document is written. The majority of this research has been focused on agents that are robotic vehicles moving around in the world, so much of the jargon in this document will revolve around that, but it's notable that this research can provide insights outside of its directed domain.

In the robotic planning domain, teams of heterogeneous agents are often used in complex missions including, but not limited to, intelligence, surveillance and reconnaissance operations [1–3]. Successful mission operations require precise coordination throughout a fleet of agents. For small teams of agents, humans are relatively good at coordinating behaviors given limited autonomy [4]. As the sizes of these teams grow, the difficulty of solving the planning problem grows dramatically. This motivates the development of autonomous algorithms that are capable of allocating resources in an intelligent way. The process of automating the planning problem frees human operators up to pay closer attention to the things that they do well: interpretation of information and overall strategy specification and oversight [4]. All autonomous algorithms in the foreseeable future will require humans to specify some amount of problem specific information that quantizes the relationship between the different mission objectives and the rigidity of constraints. From this point, autonomous algorithms can take this information and turn it into a plan that agents can execute efficiently.

1.1.1 Definitions

This section will introduce some of the basic vocabulary used in the decentralized planning community. Many of these terms mean something very specific in our field so this section will formalize some ambiguous terms. More advanced readers may skip this section as these are only the most basic definitions for the field of decentralized task allocation.

Task allocations algorithms were introduced as a tool to allocate resources during complex **missions** with a possibly non-trivial set of goals. These missions may require significant resources to complete, so the objective of task allocation algorithms is to

distribute available resources as efficiently as possible. As a computational tool, these missions are often broken up into pieces called **tasks**. Tasks can take many forms but on their most basic level they specify an indivisible objective of a mission. In more advanced mission scenarios we might also introduce **task constraints** between the tasks that specify some interaction between a pair of tasks. More formally we can redefine a mission as the Cartesian product of a set of tasks and a set of constraints between these tasks.

We use the term **agent** to refer to an entity that has capabilities to service individual tasks (or sets of tasks.) Often multiple agents are able to coordinate on which tasks of a mission they are most suitable to service. The research in this thesis was developed primarily thinking about agents as unmanned aerial vehicles (UAVs), but in general, any entity with the ability to service a task is what we define as an agent. In other related research, these agents may take the form of unmanned ground vehicles (UGVs), unmanned sea vehicles (USVs), any human operated vehicle, or even human operators sitting in front of a mission control terminal.

In order for fleets of agents to coordinate their efforts, there must be some interaction between the individual agents. All of this information that is shared between the agents during a mission is called **communication**. As would be expected, holding all other things constant, less communication is better. However, given the fact that these algorithms operate in dynamic environments, there will always be a minimum amount of communication needed to effectively allocate tasks to a fleet of agents. A common tool to evaluate how different an algorithm performs is to count the number of **messages** shared. (A parallel way to measure communication is to talk about bandwidth. The problem with using bandwidth as a measure for task allocation algorithms is that we have to worry about how messages are encoded, error correction, encryption methods, and other complexities to really talk about bandwidth. For this reason it's just more straight forward to count messages and assume that this is roughly proportional to bandwidth.) Messages in this thesis are defined as the number of outgoing broadcasts from a given agent. This implicitly assumes that it's possible for multiple agents to hear the messages broadcast from a single agent.

An agent's current knowledge of the world plus knowledge of all other agents' states is referred to as **situational awareness**. In practice, the situational awareness of each agent in a fleet will be different and is the driving force behind why coordination is needed between the agents (and thus why the agents usually cannot plan independently of each other.) The potential uses for varying degrees of this situational awareness will be discussed throughout the thesis.

The term **consensus** refers to the process by which agents come into agreement over variables relevant to the problem. In this thesis we will specifically address two different kinds of consensus. The first is **task consensus**, which is the consensus process where agents specifically agree on what agents will be performing which tasks. The other type of consensus discussed in this thesis is referred to as **situational awareness consensus**. In some of the literature, situational awareness consensus has an exact definition, but in this thesis we will just use it to refer to the consensus process for all the variables not defined in the task consensus solution.

When evaluating how an algorithm will perform in an environment we talk about the algorithm's **convergence** by specifying how long it will take until the algorithm has completed all of the relevant computation for creating the task assignment. Convergence can be difficult (or in some cases impossible) to determine in general networks and more on this will be addressed later in the thesis.

The following 5 terms describe different modes of algorithmic operation. They will be defined briefly here but a more involved and complete discussion will be introduced later in this chapter.

An important aspect of a task allocation algorithm is where the plans are computed. When we say that the algorithm is **centralized**, we are saying that all computation relevant to task allocation is being done on one machine and is then passed back out to all the agents after the allocation is complete. These algorithms are often implemented as *centralized parallel* and are often quite fast because they can utilize the fact that modern processors can compute multiple threads at the same time. When the computational power of multiple machines is used and the connection between the machines is assumed to be *strong*, we call these algorithms **distributed**.

The definition of strong is fairly loose but generally refers to reliable communication without message dropping and relatively low latency. If the communication between these machines is assumed to be *weak* or unreliable we typically refer to the resulting algorithms as **decentralized**. This distinction is important because in much of the literature of this field, distributed and decentralized are treated as synonyms but in this thesis we will define them to be different and explore how the different assumptions propagate into the resulting algorithms designed to operate in each respective communication environment.

Another important aspect of algorithmic operation can be described with the term **synchronous**. When we talk about synchronous algorithmic operations we are specifically referring to constraints placed on the relative timing of computational events in the execution of this algorithm. These constraints can be both intra-agent and inter-agent. In most iterative algorithms, parallel structures need to be synchronized every iteration to maintain the proper flow of the algorithm. In some cases this synchronization may have side effects that are undesirable or synchronization may actually turn out to be impossible. For this type of behavior we refer to resulting algorithms as **asynchronous**. Asynchronous timing does not have hard constraints between the timing of events and utilizing asynchronous computation can be a very powerful tool, especially in decentralized applications.

1.1.2 Types of Planning Algorithms

There are many important decisions to be made when deciding what type of planning algorithm to use. This first decision is usually to accept that the planner will have to be an approximate answer to the problem statement. Sometimes mission designers talk about "optimal" task allocation. Most of the time this goal is inadvisable because 1) for any significant problem size, optimality is computationally infeasible even in the most efficient computation structures and 2) the allocation will only be optimal with respect to the cost function created and the situational awareness available, often times these cost functions are approximate and being optimal to an approximate cost function is by definition an approximate solution. Because of this, most planning

problems are solved using approximate methods. These solutions vary from approximate Mixed Integer Linear Program solutions, to game theoretic optimization, to greedy based approaches. While all of these methods are valid solutions to certain problem statements, this thesis specifically focuses on greedy based algorithms. It will become clear all of the useful reasons behind this, but that is not to say that other solutions are not worth attention for specific problem domains.

Another significant decision to make is where the computation will actually happen. The jargon describing the computational methodology is often very loose in the literature so this section will attempt to create 3 specific paradigms for where computation can be executed, 1) centralized, 2) distributed, and 3) decentralized. In addition to these 3 computation structures, the designer also needs to choose when information is shared and computed: these protocols can range from forced synchronous to partially asynchronous to totally asynchronous.

Centralized

A centralized computation model refers to when the algorithm is running on a single machine, and information passing between different components of the algorithm is done through shared memory. Fast centralized algorithms are often implemented using parallelized computation structures and can take advantage of the large number of cores in modern computer systems. An important note here is that centralized implementations have an unfair stigma of being slow. From a computation perspective this is absolutely false, in fact, in most environments the truth is that these algorithms can be quite fast. For algorithms that are significantly parallelizable, computation can be fast because all pieces can have access to the current global algorithmic state almost instantaneously. This means that there is very little communication cost between the modules unlike the other paradigms listed below. Another common argument against centralized algorithms is that they create a single point of failure. This also has nothing to do with a good centralized algorithm, and since it only requires computational hardware, the computation can happen anywhere, (2 or 3 backup sites can be dedicated) and we no longer have a single point of failure, but

we still retain the properties of a centralized algorithm.

There are significant drawbacks to centralized approaches in some scenarios and these are listed below.

1. Large amounts of data in general may need to be transmitted to the centralized solver's location, whereas the processing of the data could have occurred remotely to identify the "usefulness" of information before it is sent. This remote computation can involve anything from processing video feeds or images to creating localized task allocations. Once relevant decisions are being made elsewhere we cannot really consider the planner as centralized any more.

2. The solution speed of centralized solvers are throttled by the rate at which the important information reaches the computation structure(s). If communications are slow, unreliable, or expensive, it makes sense to do as much computation as possible before moving important information through these communication links.

3. If all (or most) information needed by each agent to create its task allocation is local, much faster reaction times can be obtained by keeping all of the computation local and sharing only the results with other agents through communication.

4. In an abstract sense, distributing the computation on multiple machines might also be preferred if there are very different types of calculations that need to be performed. If, for example, the computation has large portions that can be sped up using a different computational structure such as a GPU, it would be desirable to let some computation happen in more optimized locations for certain calculations.

If the above criteria are not a concern for a particular application, centralized parallel algorithms are likely a great computational choice.

Distributed

Distributed systems can be implemented on a single machine or on separate machines. What really defines distributed algorithms is the fundamental assumption of reliable message passing between the distributed modules. In a distributed system this communication between the agents can be either passed over a network or implemented as internal memory passing, but the aspect that defines the algorithm as distributed computation as opposed to centralized computation is that it utilizes separate memory partitions between the distributed pieces. This introduces an extra layer of complexity over centralized solutions because when information about distributed pieces of the algorithm is needed, it must be transmitted through a communication channel.

To hammer this point home a bit more, the defining characteristic of algorithms that are designed to operate in distributed environment is the guarantee of strong connections. When we say strong connections we mean that each distributed node knows which other modules they are communicating with, how to get a hold of them, low message latency between each of these nodes, and that messages are guaranteed to arrive reliably. These assumptions propagate all of the assumptions of synchronization and information sharing capabilities that permeate the algorithms. A perfect application of a distributed algorithm is if domain requirements are such that a single machine does not have the capabilities necessary, one would probably move to a distributed system, that utilizes some number of machines communicating through messages in a locally, reliable way. The trade-off made when distributing computation in this way, is that time lost in the message communication must be offset by the extra computation available by utilizing multiple machines. Distributed systems also shine when information is being acquired separately and external processing can be utilized before meta-data is transmitted to other modules. (In the task assignment problem this can involve agents observing local events, then changing their plan based on this new information, then reliably communicating this information to the rest of the distributed modules.) Again, the most important aspect of these distributed algorithms is that they rely on stable communications and depend on getting information from

other modules reliably. If communication links are non-ideal then serious issues arise under this paradigm, and decentralized algorithms may be more appropriate.

Decentralized

Decentralized algorithms in general can be implemented on a single machine or multiple, but if on a single machine, the algorithm will likely be taking a performance hit. These systems are most useful in multi-module systems where communications are done exclusively via messages. Decentralized algorithms are designed for environments where there are no constraints placed on message delays, network connectivity, program execution rates, and message arrival reliability. This means that decentralized algorithms are the most robust to catastrophic changes in the communication environment. The price of this robustness may be conservative performance when communication conditions are actually favorable, but for truly decentralized environments, this trade-off is absolutely necessary. Fully decentralized algorithms are often necessary when it is desirable to have a large degree of autonomy on remote agents. Given that communications may be weak with other agents, decentralized algorithms can allow large fleets to interact without bogging the entire network down in constricting infrastructure.

1.1.3 Synchronous vs Asynchronous

In much of the task allocation world parallelized computation is used heavily. This parallelization can be happening on a single machine in a centralized architecture or on several machines in distributed and decentralized architectures. The algorithmic designer now has an extra degree of freedom to choose; when computation happens and what assumptions it makes during this computation.

Synchronous

When we think about *synchronous* computation, we are thinking about a fairly rigid structure that defines when certain information can be computed. Often synchoniza-

tion takes the form of holding off computation until a certain event happens, then releasing a set of threads to perform computation simultaneously. This idea is heavily utilized in iterative algorithms. Typically a set of agents make parallel decisions then come back together to agree on whose choices were best. This notion of modules waiting and releasing computation decisions based on the state of other computational structures is an example of synchronization. It can be a very powerful tool in many cases because it enables the algorithm to make assumptions about the information state of other modules (because, for example, they may have just agreed upon state variables). Unfortunately, in most cases it takes significant effort to enforce this synchronous behavior, often requiring that non-local constraints be placed on the algorithm's execution to force this synchronization. In the literature, this is often referred to as the synchronization penalty[5]. For computation threads running on a single machine this is usually not a significant slowdown to performance, but when information is being shared across a network, and algorithms spend time waiting for messages to arrive from physically separated machines, this penalty can become severe.

Asynchronous

The other end of the spectrum is to make the parallelized pieces of the algorithm run utilizing *asynchronous* computation. This is typically useful when the separate modules of the algorithm can run relatively independent of one another. This allows each of the modules to run its computation on its own schedule. It works well in decentralized algorithms because as a result of communication never being guaranteed, the algorithm can utilize information whenever available and not on a rigid schedule. There are some performance hits that are implied by asynchronous computation because assumptions about the information state of other agents break down. The information state of all the agents in the network is not known like it would have been had all of the agents communicated at the end of a synchronization effort. This introduces a fundamental trade-off between the rigidity of forcing an algorithm to run on a synchronous schedule versus allowing the algorithm to run asynchronously

but take a performance hit for losing information assumptions about the state of the other modules.

1.1.4 Performance Metrics

Once the computational structures of the algorithm are chosen, there are several metrics that the designer will care about when the program is executed. Depending on the relative importance of each of the following metrics, different types of algorithms may be preferred.

- *Score performance* specifically refers to how important it is that the final solution optimizes the global cost function. There can be cases where there is a lot of coupling in the score so it is a desired trait for the algorithm to be a good task allocator in terms of score, as opposed to, in other scenarios it's easy to get close to optimal to other considerations may be more important.

- *Run time* specifically refers to how much computational time the entire algorithms takes to complete, as will be contrasted below, this is typically thought of as a global measure for the entire fleet. For off-line solutions, acceptable measures of run time can be in the hours. When algorithms are run in the field, run time is typically thought of in the seconds range. Depending on the environment, the requirements for run time can be different.

- *Convergence detection* can be a difficult thing to quantify in general. It is usually trivial to post process when convergence occurred, but in some cases (especially decentralized implementations) it may be difficult to determine if the algorithm has actually converged. This leads to different definitions of what convergence should be. Does the algorithm require global convergence? Does it just require local convergence (only the individual module)? Does the algorithm only require partial convergence (in some cases being sure about the next task to be executed is enough)? The answer to this question will change what type of information is communicated as well as how much total communication is actually needed.

- *Convergence time* is typically identical to run time in global convergence metrics but in local convergence or partial convergence metrics it measures the time until this convergence detection. With this number typically smaller is better, and the smaller this number is, the faster the planner can react to dynamic changes in the environment. Because of this, in a fairly static environment, sacrifices may be made in terms of convergence time. Conversely, in very dynamic environments, convergence time may be one of the most important metrics of the algorithm.

- *Reaction time to situational awareness* is closely related to convergence time but is slightly different. It looks specifically at the turn around between situational awareness changes and how fast this information can make it into the plan cycle. In some algorithms, you cannot add information in mid-convergence (while others you can.) Different choices about reaction time will introduce different types of algorithms that are good at handling them.

Given the relative importance of these metrics, one can augment the task allocator design to make trade-offs over which criteria are more important. It will be introduced more formally in the next few chapters but in this thesis we are looking to develop an asynchronous decentralized task allocation algorithm. We are partly concerned with score, but given the communication constraints, we accept that we shouldn't be too picky over score. The algorithm will be run in real time in the field so we are looking at convergence times in the seconds. Since we are operating in decentralized environments (where decentralized is defined in the previous section) we can really only hope to hold on to local convergence. Global convergence would be nice, but we will show in a later chapter that in decentralized algorithms, it may actually be impossible to reach global consensus. The last goal is to hope to react as fast as possible to changes in situational awareness without sacrificing our convergence time constraints.

1.2 Literature Review

Centralized planners, which rely on agents communicating their state to a central server, are useful since they place much of the heavy processing requirements safely on the ground, making robots smaller and cheaper to build [6–12]. Ideally, the communication links between all elements of the system (command station, autonomous vehicles, manned vehicles, etc.) are high bandwidth, low latency, low cost, and highly reliable. However, even the most modern communication infrastructures do not possess all of these characteristics. If the inter-agent communication mechanism has a more favorable combination of these characteristics compared to agent-to-base communication, then a distributed planning architecture offers performance and robustness advantages. In particular, response times to changes in situational awareness can be made significantly faster via distributed control than those achieved under a purely centralized planner. As a result, distributed planning methods which eliminate the need for a central server have been explored [13–16]. Many of these methods often assume perfect communication links with infinite bandwidth in order to ensure that agents have the same situational awareness before planning. In the presence of inconsistencies in situational awareness, these distributed tasking algorithms can be augmented with consensus algorithms [17–26] to converge on a consistent state before performing the task allocation. Although consensus algorithms guarantee convergence on information, they may take a significant amount of time and often require transmitting large amounts of data [27].

Other popular task allocation methods involve using distributed auction algorithms [28–31], which have been shown to efficiently produce sub-optimal solutions. One such algorithm is the Consensus-Based Bundle Algorithm (CBBA) [32–34], a multi-assignment distributed auction approach with a consensus protocol that guarantees a conflict-free solution despite possible inconsistencies in situational awareness. The baseline CBBA is guaranteed to achieve at least 50% optimality [32], although empirically its performance is usually better given cost functions that are relevant to mobile robotics [35]. The bidding process runs in polynomial time, demonstrating

good scalability with increasing numbers of agents and tasks, making it well suited to real-time dynamic environments with solid communication links. Although the CBBA algorithm allows for asynchronous bidding at each iteration of the algorithm, the consensus phase relies on synchronized communication between all agents. In order to operate in a decentralized *synchronous* environment, artificial delays and extra communication must be built into algorithms to ensure agents remain in sync. These delays and extra messages reduce mission performance and may even be unrealisable in physical systems. This problem has been explored in [36] where the authors recognized that synchronization is not always feasible in distributed/decentralized applications. Although they are looking at the typical problem in the consensus literature of purely a parameter value, they identify the same questions we have to answer in task allocation: When can we safely say a state is converged? How do we construct the protocol to reduce convergence time as much as possible? Can we even guarantee that consensus is reachable? That last question is important and is handled in this thesis, if we provably can't reach consensus, what is the next step?

Extending these ideas, there has been significant work in the field of linear asynchronous consensus [17, 37–39]. The asynchronous decentralized task allocation problem is further complicated because the consensus state spaces are often extremely non-linear. Unfortunately because of this, much of the theoretical work in terms of convergence analysis breaks down. The asynchronous task assignment problem has also been approached in other domains, such as game theory [40]. Unfortunately the nature of the some of the guarantees in this work rely on continuous, convex and twice differentiable score functions. None of these properties hold for the generalized cost functions we hope to capture in the algorithms presented in this thesis. Other works have helped to identify the added complexity introduced by working with asynchronous environments [41]. Ref. [42] introduces that it is often desirable to remove all global mechanism in decentralized environments. In [43, 44] the authors used the powerful ideas of decentralized control to assign single tasks to agents in a greedy way. Their paper highlights the power of using local behaviors to govern a fleet globally with task allocation. Unfortunately the structures used in these papers

is only applicable to the matching problem, which is different than the multi-task assignment problem and not directly applicable because of the added complexity of allocating multiple tasks to each agent.

The work in this thesis extends CBBA for networked agents communicating through a distributed, asynchronous channel. The algorithmic extensions proposed in this paper allow the Asynchronous Consensus-Based Bundle Algorithm (ACBBA) to operate in real-time dynamic tasking environments with a poor communication network. As previous work has shown [45], ACBBA can provide the task space convergence properties of nominal CBBA [32], while maintaining a relatively low message bandwidth. The power of asynchronous algorithms emerges when distributed methods are implemented in real time. ACBBA applies a set of local deconfliction rules that do not require access to the global information state, consistently handle out-of-order messages and detect redundant information.

1.3 Thesis Contributions

The main contributions of this thesis are as follows:

1. **(Chapter 1)** Provided consistent and explicit classification of the centralized, distributed, decentralized, synchronous and asynchronous algorithmic domains when applied to task allocation.

2. **(Chapter 2)**Detailed task allocation problem specification for decentralized algorithms.

3. **(Chapter 3)**Provided motivation behind choices of task consensus, bundle construction and task bidding in distributed and decentralized task allocation.

4. **(Chapter 4)**Developed an algorithm called the asynchronous consensus based bundle algorithm (ACBBA) and a discussion of how the algorithm performs as a decentralized task allocation algorithm.

5. **(Chapter 4)**Defined and demonstrated the importance of local convergence in decentralized applications.

Chapter 2

Task Assignment Problem

2.1 General Problem Statement

Given a set of N_t tasks and a set of N_a agents, the goal of a task allocation algorithm is to find an allocation that maximizes a global reward function while enforcing all of the problem and domain specific constraints.

This generalized task assignment problem can be written as:

$$\begin{aligned} \underset{\mathbf{s},\mathbf{x},\boldsymbol{\tau}}{\text{argmax}} \quad & \sum_{j=1}^{N_t} R_j\left(\mathbf{s},\mathbf{x},\boldsymbol{\tau}\right) \\ \text{subject to:} \quad & \mathbf{s} \in \mathcal{S}(\mathbf{x},\mathbf{t}) \\ & \mathbf{x} \in \mathcal{X} \\ & \boldsymbol{\tau} \in \mathcal{T} \end{aligned} \tag{2.1}$$

In formulation 2.1, $R_j(\mathbf{s},\mathbf{x},\boldsymbol{\tau})$ represents the score achieved by the fleet for task j. This score is a function of $\mathbf{s}$, representing the state trajectories for all agents; of $\mathbf{x}$, a matrix of decision variables (specifically where x_{ij} is a binary decision variable equal to 1 if task j is assigned to agent i and 0 otherwise); and τ_{ij} is a variable representing the time at which agent i services task j if $x_{ij} = 1$ and is undefined otherwise. The index set that iterates over agents i, is defined as $\mathcal{I} \triangleq \{1,\ldots,N_a\}$ and the index set that iterates over tasks j, is defined as $\mathcal{J} \triangleq \{1,\ldots,N_t\}$. $\mathbf{S}(\mathbf{x},\boldsymbol{\tau})$ defines the

set of possible trajectories that satisfy both vehicle and environmental constraints as a function of $\mathbf{x}$ and $\boldsymbol{\tau}$, $\mathcal{X}$ defines the set of all feasible task allocations taking into account agent capability constraints and constraints between tasks, and $\mathcal{T}$ defines the set of all feasible task servicing times, taking into account possibly complicated temporal constraints between these tasks. In general, the full constraint space of the task allocation environment is defined by the triple of $(\mathbf{S} \times \mathcal{X} \times \mathcal{T})$. This state space has the power to specify most task allocation and motion planning problems. Because of this extreme generality, the size of the state space is uncountably infinite and thus is a very difficult space to search, even approximately. In this thesis, a series of simplifying assumptions are made to make this space searchable. Each of these assumptions will be outlined below.

Task Servicing Uniqueness

One of the first assumptions made in the task allocation literature is that every task may be assigned to at most one agent. Roughly speaking this means that two agents are not allowed to service the same task. This assumption dramatically reduces the cardinality of $\mathbf{X}$. This task servicing uniqueness constraint is formalized below as equation 2.2.

$$\sum_{i=1}^{N_a} x_{ij} \leq 1 \tag{2.2}$$
$$x_{ij} \in \{0,1\}, \forall (i,j) \in \mathcal{I} \times \mathcal{J}$$

Even though this constraint explicitly prohibits multiple agents from performing the same task, it still allows for cooperation given some creative task construction. Some approaches like posting multiple tasks at the same location and time (implying some sort of cooperation will take place) are a brute force way to establish cooperation at a task location. Other more explicit approaches were introduced by Whitten in [46]. His approach was to create sets of tasks with constraints between them and, by enforcing these constraints, he enables cooperation. Approaches using constraints between tasks will not explicitly be addressed in this thesis, but none of the assumptions made in

the problem statement presented in this chapter prohibit Whitten's unique constraint structures.

Task Independence

Another simplifying assumption is that the score achieved for each task, is independent of the completion of all other tasks and locations of all other agents. This allows us to significantly decouple the reward function so that the only two things that affect the score of the task are: the capabilities of the agent servicing the task, and what time the task is actually serviced. The form of the reward function in equation 2.1 then simplifies below in equation 2.3.

$$\sum_{j=1}^{N_t} R_j\left(\mathbf{s}, \mathbf{x}, \boldsymbol{\tau}\right) \rightarrow \sum_{j=1}^{N_t} R_j\left(\mathbf{x}_j, \tau_j\right) \tag{2.3}$$

where $\mathbf{x}_j \triangleq \{x_{1j}, \ldots, x_{N_a j}\}$ is a vector indicating which agent i (if any) will be servicing task j, and $\boldsymbol{\tau}_j \triangleq \{\tau_{1j}, \ldots, \tau_{N_a j}\}$ is the corresponding vector that indicates the service time of task j (this will have at least N_a undefined values and at most 1 defined value that specifies the service time). It is worth noting that Whitten[46] also addressed the problem of introducing more complicated task dependencies consistent with equation 2.3 above. The solution he proposed was to add constraints that make the task selection infeasible if certain constraints were not met. This introduces coupling in the task constraints instead of in the score function. Searching over the coupling in the constraints as opposed to coupling in the cost function turns out to be an easier space to search over. These additional task constraints are not explored in this thesis, but the important take away is that the above assumptions still allow for significant task coupling through constraints in environments where this is needed.

Decouple path planning solution

Another aspect of our problem formulation is that we avoid explicitly solving for $\mathbf{s}$, the actual dynamic paths that every agent must follow to service its assigned tasks. In our problem formulation, we only guarantee that given a feasible task allocation

$\mathbf{x}$, and a set of service times $\boldsymbol{\tau}$, there *exists* some $\mathbf{s} \in \mathcal{S}(\mathbf{x}, \mathbf{t})$. The problem of actually optimizing $\mathbf{s}$ is deferred to the field of path planning. Typically a conservative estimate of the maneuvering capabilities of the agent are used and Euclidean paths are assumed. In some of our recent work we have looked a bit more closely at the possible trajectories for $\mathbf{s}$ for better estimates of $\boldsymbol{\tau}_j$, but we still only must guarantee that a feasible trajectory exists.

2.1.1 Full problem description

At this point we have completed the description of our problem statement as shown below in equation 2.4.

$$\begin{aligned} \underset{\mathbf{x},\boldsymbol{\tau}}{\text{argmax}} \quad & \sum_{j=1}^{N_t} R_j\left(\mathbf{x}_j, t_j\right) \\ \text{subject to:} \quad & \sum_{i=1}^{N_a} x_{ij} \leq 1, \forall i \in \mathcal{I} \\ & \mathbf{s} \in \mathcal{S}(\mathbf{x}, \mathbf{t}) \\ & \mathbf{x} \in \mathcal{X} \\ & \boldsymbol{\tau} \in \mathcal{T} \\ & x_{ij} \in \{0,1\}, \forall (i,j) \in \mathcal{I} \times \mathcal{J} \end{aligned} \tag{2.4}$$

2.1.2 An alternate algorithmic friendly problem statement

This formulation, although complete mathematically, still isn't structured in a convenient form to explain the algorithmic tools we use to solve it. The following section will introduce a bit more notation to illuminate where the problem can be decoupled, what pieces can be solved separately, and introduce some of the data structures used to keep track of the relevant information.

1. For each agent we define an ordered data structure called a *path*, $\mathbf{p}_i \triangleq \{p_{i1}, \ldots, p_{i|\mathbf{p}_i|}\}$, whose elements are defined by $p_{in} \in \mathcal{J}$ for $n = 1, \ldots, |\mathbf{p}_i|$. The path contains the information representing what tasks are assigned to agent i. Their order

in the path data structure represents the relative order in which they will be serviced.

2. The score function matrix $\mathbf{c}(t)$ is constructed of elements c_{ij} that represent the score that agent i would receive by servicing task j at time t.

3. For each agent i we define a maximum path length L_i representing the maximum number of tasks that the agent may be assigned. This is an artificial constraint placed on the problem to help with computation and communication efficiency. As will be discussed with the algorithms later, this constraint is often paired with an implicit receding time horizon to guarantee an agent doesn't commit to tasks too far in the future and exhaust its bundle length.

Given these new definitions we may augment the problem definition as follows:

$$\begin{aligned}
\underset{\mathbf{x},\boldsymbol{\tau}}{\text{argmax}} \quad & \sum_{i=1}^{N_a}\left(\sum_{j=1}^{N_t} c_{ij}(\tau_{ij}(\mathbf{p}_i(\mathbf{x}_i)))x_{ij}\right) \\
\text{subject to:} \quad & \sum_{j=1}^{N_t} x_{ij} \leq L_i,\ \forall i \in \mathcal{I} \\
& \sum_{i=1}^{N_a} x_{ij} \leq 1,\ \forall j \in \mathcal{J} \\
& \mathbf{s} \in \mathcal{S}(\mathbf{x}, \mathbf{t}) \\
& \mathbf{x} \in \mathcal{X} \\
& \boldsymbol{\tau} \in \mathcal{T} \\
& x_{ij} \in \{0,1\},\ \forall (i,j) \in \mathcal{I} \times \mathcal{J}
\end{aligned} \tag{2.5}$$

Given this new formulation specified in equation 2.5, we start to see how we can distribute computations across the agents. At this point, given $\mathbf{x}$, the matrix $\boldsymbol{\tau}$ can be determined by each agent independently. Furthermore, with $\mathbf{x}$, each individual agent can find its own contribution to the full matrix $\boldsymbol{\tau}$ by solving equation 2.6.

$$\max_{\tau} \quad \sum_{j=1}^{N_t} c_{ij}(\tau_{ij}(\mathbf{p}_i(\mathbf{x}_i)))x_{ij} \tag{2.6}$$

The first thing we can note about equation 2.6, is that each agent i will first have to optimize its own path $\mathbf{p}_i$ if it is given what tasks it has a responsibility to service $\mathbf{x}_i$ subject to the path planning feasibility constraint $\mathbf{s} \in \mathcal{S}(\mathbf{x}, \boldsymbol{\tau})$ and the feasibility of arrival times constraint $\boldsymbol{\tau} \in \mathcal{T}$. During the path optimization, optimal values for $\boldsymbol{\tau}$ will also be determined. Then from this optimization, the agent's individual score for each assigned task will then also be fully defined. The main takeaway from this point is that the main source of distributed coupling in the path assignment problem is restricted to the choice of the assignment vector $\mathbf{x}$. Given this, once all agents agree on a value for $\mathbf{x}$ then the distributed task assignment problem has been solved. The next two chapters in this thesis will be focused on ways of solving the problem of finding a consistent value of $\mathbf{x}$ across all agents.

2.2 Algorithmic Design Goals

The specific task allocation environment explored in this thesis places some constraints on the form of $\mathbf{c}(t)$ and also how the agents are able to interact and communicate with each other. The following section will outline the extent of the other constraints that are placed on the task allocation environment.

Score functions

In this thesis the functional form for the reward received for the completion of a task will be of the form of a time windowed exponential decay function.

$$r_{ij}(t) = \begin{cases} 0 & \text{for } t < t_{\text{start}} \\ e^{-\gamma(t - t_{\text{start}})} & \text{for } t_{\text{start}} \leq t \leq t_{\text{end}} \\ 0 & \text{for } t > t_{\text{start}} \end{cases} \tag{2.7}$$

In equation 2.7, in the task definition, we define $r_{ij}(t)$ to be the reward received for agent i completing task j at time t. We can also define intrinsic variables that specify task properties: t_{start} represents the beginning of the time window for the

task, t_{end} represents the end of the time window for the task, and γ is the discount factor for servicing the task late in the available time window.

In addition to the reward for servicing a task, we also introduce a penalty of moving around in the environment into the overall score function. This cost can take many forms, in this thesis, specifically, we just talk about the cost being the distance penalty of moving around in the environment. The physical realization of this cost is equivalent to a fuel penalty when we are dealing with ground vehicles. Unfortunately it is not a direct parallel when considering air vehicles. This is because it does not explicitly account for fuel burned while loitering.[1] Since the penalty is coupled to other tasks that an individual agent services during its assigned mission, we can understand why we have been writing the score of an individual task as a function of the entire path that the agent actually traverses. For algorithmic convergence reasons the way that this penalty is introduced into the full score function is dependent on the particular algorithmic implementation. This then creates the total score function for agent i servicing task j at time τ_{ij} that is described in equation 2.8.

$$c_{ij}(\tau_{ij}(\mathbf{p}_i(\mathbf{x}_i))) = r_{ij} - \mathbf{penalty} \tag{2.8}$$

A more formal treatment of this score function will be introduced when it is bound to a specific algorithm but equation 2.8 gives the basic form of all score functions that will be considered in this thesis.

Communication Environment

The algorithm developed later in Chapter 4 of this thesis envisions that we are operating in a truly *decentralized* environment. As a reminder, operating in decentralized environments implies that network connectivity is never guaranteed to exist between any 2 nodes; agents may enter and leave the network (through geometry or comm drop out); when agents are actually connected to each other, there are no guarantees to how often they are able to communicate, or what the message latencies will be

[1] With small vertical take off and landing we could think about landing temporarily instead of loitering and the penalty becomes much closer to an actual algorithmic penalty.

when they decide to communicate. There is also an assumption that communication is so sparse that the agents have basically no situational awareness about the other agents and only have the bandwidth to perform task consensus. Since communication is regulated so heavily, a goal of the task allocation algorithm will be to reduce the number of messages as much as possible with the information available to utilize the bandwidth as efficiently as possible. A final goal of the task algorithm operating in this communication environment is that even in the absence of interaction with any peers, agents must be able to do something reasonable.

Tasking Environment

In this thesis we assume that the tasking environment will be very dynamic. This means that not only will off-line task allocation solutions be unable to predict the conditions that the agents will encounter in the field, but these environmental conditions may change significantly during the mission execution itself. Because of this, the algorithm must be robust to handle constant changes in situational awareness in the environment. This includes significant position changes of the vehicles in the environment, teammate agents coming off-line and on-line mid-convergence, as well as tasks appearing and disappearing during the convergence process.

The constraints outlined in this chapter through the formal problem specification, and the description of the communication and task environments, all combine to create a problem statement that was not solvable before the author's proposed solution.

Chapter 3

Planning Approaches

The following section outlines the progression of algorithms that created the framework for the main result of this thesis, the *Asynchronous Consensus Based Bundle Algorithm* (ACBBA). This chapter will highlight and explain the decisions that were made during the algorithmic development process and show how these decisions shaped the overall class of algorithms that were created.

3.1 Coupled Element

As was pointed out in Chapter 2, the only coupled variable in a distributed problem statement is the assignment matrix $\mathbf{x}$. Since this coupling exists, the agents need some way of agreeing on consistent values for $\mathbf{x}$ in a distributed way. To create this consistency, there are three types of coordination techniques: 1) create an a priori partition in the task space such that the task sets each agent can service are disjoint; 2) perform situational awareness consensus (consensus on all variables that are not $\mathbf{x}$) and hope that agents independently create allocations that satisfy all of the relevant constraints on $\mathbf{x}$; or 3) directly incorporate the constraint feasibility of $\mathbf{x}$ during the convergence of the planning algorithm using communication and consensus algorithms.

1) Creating an a priori partition in the task environment effectively chops up the task space and only allows each agent to bid on a subset of the overall task set. Given

a higher-level task allocation or a human operator, it may be reasonable to use this method for small teams in relatively static environments. This is because for some environments it may be obvious what agents should be servicing which tasks. However, typically in environments with large numbers of relatively homogeneous agents, or dynamic environments it becomes non-obvious which agents can best service which tasks. Effectively by creating this partition, the algorithm is placing artificial constraints on what allocations are available and may lead to poor task assignments. Therefore, it's noted that partitioning the task space can be an effective technique in some task environments, but for the environments explored in this thesis, more elaborate techniques are needed.

2) The technique to create a consistent assignment matrix has been referred to as *implicit coordination*. The main emphasis of this method is to arrive at situational awareness consensus[1] before starting the planning algorithm, then run effectively centralized planners on each agent independently with the idea that if each agent started with the identical information then each agent will produce consistent plans. This method has been explored in the literature [17–26] and was popular as a relatively straight forward way to decentralize a task allocation algorithm. The benefit of using implicit coordination over that of task space partitioning described above is that by not limiting the assignment space a priori, more dynamic reaction to tasks in the field are enabled. Implicit coordination also has the added benefit that if the task environment is highly coupled (with many agent-to-agent and task-to-task constraints), then the plans produced are able to recognize and exploit this structure. It then becomes possible to find task allocations that include highly coordinated behaviors between the agents.

All of these benefits come with the caveat that the initial situational awareness consensus must happen before the planner can start producing plans. In order to guarantee that agents produce the same assignments $\mathbf{x}$, this situational awareness consensus process may require large amounts of bandwidth [27]. The problem here is that

[1]Situational awareness consensus is the state where all agents agree on all variables relevant to the initial conditions of the task allocation problem.

there are many types of variables involved in the situational awareness consensus process. Some variables in the situational awareness consensus problem are easy to find because the information creator is a trusted source. Variables like an agent's location or the agent capabilities can be trusted as fact when they are shared by that agent. Agreeing on more dynamic variables like task positions, or task scores may be a tougher problem because each of the agents may know different information about the world around them. To get accurate information about these more dynamic variables, it may require a significant consensus effort. The major drawback to using implicit coordination ends up being that the significant consensus effort only indirectly affects the actual constraints on $\mathbf{x}$. If the final estimates of the situational awareness variables do not converge to within arbitrarily tight bounds there is no *guarantee* of conflict free assignments.

3) The third solution completely ignores learning anything about the situational awareness of the other agents, and only requires that $\mathbf{x}$ remain consistent according to the constraints described in the problem statement (equation 2.5). The power of this solution is that all of the consensus effort is spent on maintaining a consistent value for $\mathbf{x}$. This solution is preferable if there are few inter-agent constraints and inter-task constraints (the more coupled the task environment becomes, the more difficult the task consensus problem becomes) or if the communication environment is not necessarily reliable such that it would be difficult to reach complete fleet-wide consistent situational awareness. In decentralized environments, we are often not worried about intense cooperation (just a conflict free distribution of tasks.) Given that in these environments the communication links are often non-robust, especially across larger distances, only broadcasting the information directly relating to the constraints on $\mathbf{x}$ is preferable.

There is a trade-off between implicit coordination and algorithms using task consensus. The choice is deciding if it is easier to converge on a consistent assignment vector $\mathbf{x}$, or converge to arbitrarily tight bounds on all other significant variables. Typically in our research we assume that the size and static nature of $\mathbf{x}$ is much easier to con-

verge on than the dynamic values of all the worldly variables; however, occasionally this is not the case and an implicit coordination approach may be appropriate. In this thesis we assume that situational awareness consensus would take too long in dynamic environments and thus pure implicit coordination techniques are unavailable for use.

In this thesis the third solution for handling the inter-agent constraints is used[2].

3.2 The Algorithmic Decisions

Up to this point we still have significant flexibility on how we construct each agent's assignment and how the agents are able to come to consensus on a final value for **x**. This section will help to introduce some more of the details about the class of algorithms that will be used in this thesis.

3.2.1 Plan horizon

The simplest decision is how far into the future are we going to plan. Options include, agents only deciding on their next task, their next n tasks, some planning time horizon, some combination of n tasks and time, etc. We typically implement a combination of n tasks and a time horizon. However, each can be useful in its own environment, so the section below will describe the fundamental trade-offs that need to be made.

• If the plan horizon is only to bid on the *next task* for each agent to perform simple planning algorithms are available for use. Many of the distributed planning complexities are introduced when agents are assigned multiple tasks simultaneously. Algorithms that only need to assign a single task to each agent will also have relatively short convergence times because there is relatively little information that needs to be

[2]Appendix A of this thesis discusses the idea of merging all three solutions in order to retain the complexity reduction of a priori partitioning, and the convergence robustness of the task consensus, while gaining the increase in performance in highly coupled environments that implicit coordination allows.

agreed upon, compared to other approaches. The problems with this type of planning approach lie in performance of the actual fleet at plan execution time. The most basic trouble with only bidding on the first task is that it may be impossible to pick what the next best thing to do is without knowing what everyone else is going to do, well into the future. In worst case, the resulting mission score can become arbitrarily bad if agents are only able to look at the next best task into the future. An example of when this worst case behavior can arise is if there is a fast agent that can get some small non-zero score for performing tasks; this agent could finish a large number of tasks before more specialized agents are available. In this scenario all of the more specialized tasks will have the have completed poorly and the mission score would end up being much lower than it could have been. Although this is an extreme example, lesser degrees of this worst case behavior will often happen when agents move around in the world in a strict greedy sense. Also, this type of architecture can create situations where every time a task is completed, a new planning exercise will commence, and agents en-route to their next task may be outbid and will have spent significant amounts of time travelling to tasks they will not end up servicing. The overall behavior of these planners in general will seem very short-sighted and it will not be lead to the desired performance for most environments.

• If the algorithmic choice is to bid in the *next n tasks*, much more intelligent looking plans can be produced than when $n = 1$. Planners that assign multiple tasks to each agent tend to be much more complicated because typically these extra assignments introduces more coupling in the task environment. If an agent is responsible for servicing multiple tasks, both the scores on these tasks and the feasibility of actually completing them are highly dependent on which other tasks the agent is assigned. In this thesis we call these sets that each agent plans on servicing a *bundle*. This bundle is ordered such that the tasks that are chosen first are placed earlier in the bundle. In this structure we are able to preserve a hierarchy of dependence for the scores (and feasibility constraints) because the score for each task will only depend on tasks located earlier in the bundle. An important thing that we have to watch

out for with this type of planner is to not plan too far into the future, such that some tasks go unassigned while some agents idle, waiting for higher value tasks in the future to become active. This can easily happen if there are are time periods where large numbers of high value tasks available surrounded by other significant time with only low value tasks.

• The next possible planning methodology would be to only bid on tasks within some *time horizon*. This ends up fixing the problem of planning on servicing tasks that are unnecessarily far into the future and allowing more immediate tasks to go unserviced. One problem that arises from planning with a time horizon is that the length of the planning horizon needs to be tuned to be long enough to capture relevant aggregate behaviors from the agents' plans without planning too far into the future such that the planning problem is unnecessarily complex. Other problems with this solution is that it is possible for agents to have no assignment during the specified time horizon, but it may make sense for agents to start travelling to a task "hub". This myopic behavior can produce catastrophic results if it takes agents longer than the planning horizon to travel between tasking areas. The resulting behavior will be that agents will never end up adding tasks to their bundles.

• Typically, the combination of using a time horizon and an n task look ahead is preferred. The two strategies end up complimenting each other well. The introduction of the n task look ahead makes tuning the plan horizon much simpler. This is because since only a maximum of n tasks will be added, an over estimate of this time horizon can be used without fear of creating too much computational complexity. There is some art in choosing the time horizon and n and they will usually be problem specific, but by considering both algorithms can produce desirable plans over a large spectrum of missions.

3.2.2 Bundles vs. Tasks

Given the discussion above, the algorithms produced in this thesis will be assigning multiple tasks per agent. With this decision, there are two choices on how to handle

multiple tasks: 1) bids can be made on entire bundles at a time (recall that a bundle is an ordered vector of all the tasks assigned a given agent), or 2) bids can be placed on individual tasks then bundles can be built sequentially one task at a time.

1) First consider looking at bidding on entire bundles of tasks. Given the problem formulation in Chapter 2 there may be $\binom{N_t}{L_i} L_i!$ possible bundles for the agents to choose from, where as a reminder, L_i is the maximum allowable length for a bundle. Enumerating all possible bundles has been explored significantly in the literature [47–49]. Bidding on bundles will usually result in a higher score because the agents are able to explicitly account for the unavoidable coupling that exists between tasks serviced by the same agent. This includes everything from travel time to clusters of similar tasks to coupling between the tasks themselves (although this thesis does not consider explicit task coupling, it is a very relevant problem in some domains.)

The downside to bidding on entire bundles though is that it makes for an incredibly complicated consensus space. The space then consists of every possible bundle where the actual feasible space becomes extremely sparse because any two bundles with just one of the same tasks becomes infeasible. Both the size of the space and the number of constraints grows exponentially in the size of the bundle length L_t. Although this space still remains finite and is computable, the problem becomes exponentially harder to both solve and perform consensus on and its generally preferable to avoid this extra complication.

2) The natural result is then to bid on the tasks independently. This still allows for the decision of either constructing entire bundles all at once or sequentially adding tasks to construct bundles. If bundles are created all at once then there is an extra bit of effort in parsing up the bids on each individual task. This can be difficult at times because in general there can be significant coupling that exists between the tasks so it may be difficult accurately partition the bundle score. When adding a single task at a time, it is very easy to assign scores to each marginal contribution for each task, but it also might produced suboptimal bundles because of the myopic nature of greedily adding one task at a time. The trade-offs between these two methods are outlined in

the section below.

3.2.3 Assigning Scores to Tasks

The class of algorithms addressed in this thesis has now been narrowed to consider iteration through task consensus, assigning tasks to agents for only n tasks at a time and not looking past some time horizon, then making these bids on individual tasks instead of bundles as a whole. There next algorithmic refinement is to decided if bundles should be built all at once then decide the value of each task in the bundle, or if bundles should be built up incrementally, one task at a time.

1) The first possible implementation involves constructing bundles all at once then assigning scores to each task, identifying how much each task contributed to the score of the overall bundle. This type of structure can explicitly account for all of the coupling between tasks in a given agent's bundle (especially travel times and favorable environmental geometries.) This method, however, has a few difficulties. The first one is determining how to actually assign a score to each individual task. This is a difficult problem because the marginal score for adding a task to a bundle can depend on every other task that was previously in the bundle. The problem then becomes how to assign values to each of these tasks given the strong coupling between them. Should a task that increases the marginal score of every other task in the bundle be allowed to share some of the score that it is enabling? The second difficulty with this approach is the total number of bundles that can be considered.The number of possible bundles at each iteration with this method grows as $\theta(N_a\binom{N_t}{L_i})$. The third major issue with this formulation arises by creating task scores that are completely dependent on the rest of the tasks in the bundle also being assigned. This makes the consensus problem much more difficult because very few bids end up staying the same between iterations. It leads to much more chaos during the consensus problem which decreases the rate of convergence.

2) An alternative to the above approach is to build up bundles incrementally, one task at a time. In this case the score for adding a task to the bundle is simply

the incremental cost for adding that particular task to the bundle. This method is essentially a greedy search through the bundle space that ends up reducing the size of the search space to $\theta(N_a N_t L_i)$. Another benefit of this search space is that clear dependencies are defined for which bids depend on each other. A bid added to the bundle is only dependent on bids that were already located in the bundle, which means that the value of a bid is independent of all of the bids made after it. What this ends up doing is that it adds some stability to the consensus space because bids have reduced their number of dependencies, and thus convergence rates increase.

This section has further defined which types of algorithms will be addressed in this thesis. These decisions were:

1. Plan with a fairly long time horizon but only allow at most n tasks to be added to an agents bundle at a time.

2. The bids for the algorithms should be placed on tasks instead of bundles.

3. The scores for each of the tasks should be computed incrementally as bundles are built up.

3.2.4 Distributed notation

A slight complication is introduced when the problem is decentralized in this manner because it creates an environment where during the execution of the algorithm, agents may have a different view of the task consensus space from each other. For this reason we will define a few new data structures to help with keeping track of this possibly inconsistent information. These definitions will be referred to throughout this chapter.

1. A *winning agent list* $\mathbf{z}_i \triangleq \{z_{i1}, \ldots, z_{iN_t}\}$, of size N_t, where each element $z_{ij} \in \{\mathcal{I} \cup \emptyset\}$ for $j = 1, \ldots, N_t$ indicates who agent i believes is the current winner for task j. Specifically, the value in element z_{ij} is the index of the agent who is currently winning task j according to agent i, and is $z_{ij} = \emptyset$ if agent i believes that there is no current winner.

2. A *winning bid list* $\mathbf{y}_i \triangleq \{y_{i1}, \ldots, y_{iN_t}\}$, also of size N_t, where the elements $y_{ij} \in [0, \infty)$ represent the corresponding winners' bids and take the value of 0 if there is no winner for the task.

3. A vector of *decision timestamps* $\mathbf{s}_i \triangleq \{s_{i1}, \ldots, s_{iN_a}\}$, of size N_a, where each element $s_{ik} \in [0, \infty)$ for $k = 1, \ldots, N_a$ represents the time stamp of the last information update agent i received about agent k, either directly or through a neighboring agent.

4. A *bundle*, $\mathbf{b}_i \triangleq \{b_{i1}, \ldots, b_{i|\mathbf{b}_i|}\}$, of variable length whose elements are defined by $b_{in} \in \mathcal{J}$ for $n = 1, \ldots, |\mathbf{b}_i|$. The current length of the bundle is denoted by $|\mathbf{b}_i|$, which cannot exceed the maximum length L_t, and an empty bundle is represented by $\mathbf{b}_i = \emptyset$ and $|\mathbf{b}_i| = 0$. The bundle represents the tasks that agent i has selected to do, and is ordered chronologically with respect to when the tasks were added (i.e. task b_{in} was added before task $b_{i(n+1)}$).

Using the refinement of the planning problem introduced in the previous two sections, the simplest task allocation framework that attempts to solve this problem, called the *sequential greedy algorithm*, will be introduced. This algorithm has been called many other things in the literature when applied in different domains but at its most basic form it is a centralized greedy auction.

3.3 Sequential Greedy

Given our problem formulation up to this point we have decided that we want to decouple the computation into individual agents deciding what tasks optimize their personal score, then use a consensus protocol to guarantee consistency in the $\mathbf{x}$ constraints. In this environment, there are still 2 key decisions to make: 1) how are each of the agents going to decide which tasks they would like to perform; and 2) how do we get all agents to agree on who services each task. The first algorithm that was created for this problem is called the sequential greedy algorithm, and it solves both of these problems posed above. It will consist of an algorithm that alternates between

2 phases: 1) a bundle building phase and 2) a consensus phase. The algorithm will be described in its entirety, and then we will go back and explain what decisions were made, why they were made, and ultimately why this algorithm was not sufficient for the problem statement posed in this thesis.

3.3.1 The Algorithmic Description

Phase 1: Bundle Building Phase

The first phase of the algorithm runs on each agent independently. In this phase each agent produces a candidate *bid*, that if they win in phase 2, they will be responsible for servicing. The process begins by each agent looking through all available tasks and computing a score on each of these available tasks. The set of available tasks is defined as any task that has not been assigned by phase 2, the Consensus Phase (what it means for a task to be *assigned* will be introduced in the next section.) For each task j in this available list of tasks, each agent i computes a score, $c_{ij}(\mathbf{p}_i)$ for each task. The form for this score function will also be described below but it is a function of all of the other tasks the agent is already responsible for servicing. When a score is computed for all available tasks for a given agent, the largest score is chosen as the candidate bid that is made up of the pair of task and score. If the agent cannot find a bid with a positive score (due to capability reasons, bundle length restrictions or lack of available tasks) then we say this agent has "converged" and is just an observer for the rest of the algorithm. Every agent with a valid bid then moves to a phase 2, if 0 agents have been able to create a bid then we say that the algorithm has converged.

Phase 2: Consensus Phase

At the start of the consensus phase all of the agents have created a bid and have shared these bids with each other. This sharing can be done through a centralized auctioneer or it can be done via a fully connected network by every agent sharing their bids with every other agent. All of the bids are then compared and the single bid with the absolute highest score is chosen as the winner. The winning task $j^\star$ is

then placed at the end of the bundle for the winning agent $i^\star$, at the appropriate location in the winning agent i's path, the jth row in the servicing time matrix is populated, and all of the agents' winning agent and bid lists are updated. In the notation presented this includes the following:

$$\begin{aligned} x_{i^\star j^\star} &= 1 \\ x_{ij^\star} &= 0, \quad \forall i \neq i^\star \\ \mathbf{b}_{i^\star} &\leftarrow (\mathbf{b}_{i^\star} \oplus_{\text{end}}, j^\star) \\ \mathbf{p}_{i^\star} &\leftarrow (\mathbf{p}_{i^\star} \oplus_{n_{j^\star}} j^\star) \\ \tau_{i^\star j^\star} &= \tau^\star_{i^\star j^\star}(\mathbf{p}_i \oplus_{n_{j^\star}} j^\star) \\ \tau_{ij^\star} &= 0, \quad \forall i \neq i^\star \\ z_{ij^\star} &= i^\star, \quad \forall i \\ y_{ij^\star} &= c_{i^\star j^\star}(\mathbf{p}^\star_i), \quad \forall i \end{aligned} \tag{3.1}$$

After this task is assigned it is removed from all of the agents' list of available tasks, and then the algorithm moves back into Phase 1.

Cost Function

Computing the score for a task is dependent on the tasks already in the agent's path (or equivalently bundle). This means that given a task j and a path for agent i, $\mathbf{p}_i$ there is an optimal projected service time for the task. This optimization consists of maximizing the incremental score of adding the task to the path, while satisfying all of the constraints imposed by equation 2.5. This process can be described in 2 main steps: 1) Agent i attempts to insert task j in between some elements in the path, and 2) given this location in the path, the agent computes the optimal service time and produces an estimate marginal score for adding this task to its path. This process is then repeated for every available location in the path and the maximum marginal score is chosen as agent i's candidate bid for task j (in the full algorithm described above in Phase 1, this candidate bid for task j is then compared with the

bids of all other available tasks and the largest of the candidate bids is then agent i's bid for that iteration). A more formal description of this process is outlined below using precise notation and language.

This process outlines the sequence of steps for agent i to compute a candidate bid on task j, given that agent i already has a path defined as $\mathbf{p}_i$ (this can be trivially applied to an empty path by letting $\mathbf{p}_i = \emptyset$).

1) Task j is "inserted" in the path at a location n_j. We will define a new notation as the pair $(\mathbf{p}_i \oplus_{n_j}, j)$, where this notation signifies inserting task j at location n_j in path $\mathbf{p}_i$. The process of inserting task j into the path at location n_j involves incrementing the locations of all of the previous path elements in $\mathbf{p}_i$ from $n_j : |\mathbf{p}_i|$ up by one and changing path element at location n_j to be task j (i.e $(\mathbf{p}_{i(n+1)} \oplus_{n_j}, j) = p_{in}, \forall n \geq n_j$ and $(\mathbf{p}_{in_j} \oplus_{n_j}, j) = j$).

2) After a candidate location in the path is chosen for inserting task j, the problem becomes choosing the optimal service time $\tau^\star_{ij}(\mathbf{p}_i \oplus_{n_j}, j)$. This notation can be read as the optimal service time for agent i of task j when task j has been inserted in the path at location n_j. This problem can be posed as the following optimization:

$$\begin{aligned} \tau^\star_{ij}(\mathbf{p}_i \oplus_{n_j}, j) = \quad & \underset{t \in [0,\infty)}{\operatorname{argmax}}\, c_{ij}(t) \\ \text{subject to:} \quad & \tau^\star_{ik}(\mathbf{p}_i \oplus_{n_j}, j) = \tau^\star_{ik}(\mathbf{p}_i), \quad \forall k \in \mathbf{p}_i \\ & \mathbf{s}_i \in \mathcal{S} \end{aligned} \tag{3.2}$$

where k is an index for all tasks already in $\mathbf{p}_i$ and $\mathbf{s}_i \in \mathcal{S}$ is the path feasibility constraint introduced in Chapter 2. Here we are optimizing the servicing time given two main constraints. The first is that we cannot change the service times for tasks that are already located in the path. This is a choice that we made with the algorithm because alternatively we could have allowed for all of the times to be shifting around. As long as the marginal score of introducing that task into the path is higher than any other consideration (including shifting the times around) it would be allowed to be the bid. We chose not to allow the shifting of times for a few reasons involving

convergence and cooperation. Without the constraint posed above in equation 3.2 the agent could choose to just not do a task in its path (by effectively shifting the completion time of it to infinity) because the task it is trying to add has a higher score. This leads to problems when agents start depending on other tasks actually being serviced when they were told they would be serviced. For this reason we sometimes see lower scores than possible but we gain bid credibility and agent accountability for servicing tasks they bid on. The second constraint is the path feasibility constraint. As was mentioned before, we are not explicitly ever solving for $\mathbf{s}_i$, but we must guarantee that a feasible $\mathbf{s}_i$ exists, and thus it can be found at a later time by more specialized algorithms. In practice this is relatively easy to do with a small buffer on vehicle speed. This allows us to conservatively estimate locations an agent can likely reach given the time constraints.

Note that the optimization defined in equation 3.2 is a continuous time problem, which, for the general case, involves a significant amount of computation. The optimal score associated with inserting the task at location n_j is then given trivially by computing $c_j(\tau_{ij}^{\star}(\mathbf{p}_i \oplus_{n_j}, j))$. This process is repeated for all $n = 1 : |\mathbf{p}_i|$ by inserting task j at every possible location in the path. The optimal location is then given by,

$$n_j{}^{\star} = \underset{n_j}{\operatorname{argmax}}\, c_j(\tau_{ij}^{\star}(\mathbf{p}_i \oplus_{n_j}, j)) \tag{3.3}$$

and the final score for task j is $c_{ij}(\mathbf{p}_i) = c_j(\tau_{ij}^{\star}(\mathbf{p}_i \oplus_{n_j{}^{\star}}, j))$. As was stated above this optimization can be difficult to solve for arbitrarily complicated cost functions, but typically in practice we use slight time decay cost functions. With this algorithm, the cost function of choice is usually of the form specified in equation 2.8. In this particular function the role of the penalty takes on the form of a fuel estimate along

the path chosen. The complete form for this cost function then becomes

$$r_{ij}(t) = \begin{cases} 0 & \text{for } t < t_{\text{start}} \\ e^{-\gamma(t-t_{\text{start}})} & \text{for } t_{\text{start}} \leq t \leq t_{\text{end}} \\ 0 & \text{for } t > t_{\text{start}} \end{cases} \tag{3.4}$$

$$c_{ij}(t) = r_{ij}(t) - f_i \left(d\left(\mathbf{p}_i \oplus_{n_j}, j\right) - d\left(\mathbf{p}_i\right) \right)$$

where f_i is the fuel cost for agent i, and d is a function computing the added length of the path traversed by agent i when task j is inserted into its path. Since the score function has an exponential decay, between any 2 tasks usually the earliest possible time is chosen (that satisfies $\mathbf{s}_i \in \mathcal{S}$) and therefore there is no need for a full optimization. With different cost functions more elaborate methods for determining this servicing time may be necessary. An example of a more complicated cost function is one proposed by Ponda [50]. If the score has any stochastic elements, picking the times becomes a robustness operation and is not as straight forward as sooner is better. In this environment, a more complete optimization is chosen for picking the servicing times.

This algorithm is guaranteed to converge in polynomial time, assuming

$$\begin{aligned} N_t &< \infty \\ \text{or} \quad N_a < \infty \quad &\text{and} \quad |L_i| < \infty \end{aligned} \tag{3.5}$$

at a rate that grows in worst case as $\Theta(\min(N_t, N_a L_i) N_t N_a L_i)$.

3.3.2 Where Sequential Greedy Fails For Our Problem Statement

The main problem with this algorithm in decentralized environments is that it takes $\Theta(\min(N_t, N_a L_i) D)$ fleet-wide synchronized iterations to converge. In this equation, D represents the network diameter (intuitively we need to add D because as the

network diameter grows, it takes more time for information to propagate throughout the network, and it slows the overall convergence down). When we are operating in environments where we do not have a very strong network architecture, reaching a fleet-wide synchronous state repeatedly is very difficult, if not impossible. The algorithm presented here cannot progress without all of the agents meeting together at each iteration to agree on the next best task. For this reason, this algorithm breaks down in large distributed architectures and any sort of decentralized architecture. In other communication structures not explicitly considered by this thesis, such as centralized, this algorithm turns out to be especially fast because the computation is very efficient. Additionally in centralized architectures there is no worry about the delays introduced from synchronizing spatially separated agents. For this reason, the sequential greedy algorithm makes a very good centralized greedy task allocation solver, but a relatively poor distributed and decentralized algorithm. The goal of the next algorithm presented is to reduce the total number of iterations the algorithm needs in order to converge, and thus allow a more efficient distribution of computation.

3.4 Consensus-Based Bundle Algorithm (CBBA)

The development of the Consensus-Based Bundle Algorithm (CBBA) was based around the idea that the sequential greedy algorithm was a good idea but it took too many synchronous iterations to converge. The observation was made that it should be possible to converge, or at least attempt to converge on multiple tasks per iteration. Out of this idea we discovered that a naive approach would not suffice and produced a non-trivial distributed extension of the sequential greedy algorithm called CBBA.

3.4.1 Algorithmic Description

Although CBBA is very similar to the sequential greedy algorithm, we will give a complete description of the algorithm, and we will attempt to highlight the location

of the differences between CBBA and the sequential greedy algorithm. CBBA retains the 2 phase structure of a bundle building phase and a consensus phase.

Phase 1: Bundle Construction

During this phase of the algorithm, each agent will be acting independently of one another. The main form of this phase will be that each agent is continuously adding tasks to its bundle in a greedy fashion until it is incapable of adding any others (either due to lack of doable tasks or reaching the bundle limit L_t). The algorithm progresses by looking through the set of available tasks(an available task is a task that is not completed and is not already in the agent's bundle), computes a score for each task then checks this score against a list of the the current winning bids. The bid is kept as a candidate next best bid if it is greater than the current best bid for that task. After this process has been completed for all of the available tasks, the agent selects the bid with the highest score and adds that bid to the end of its bundle and its appropriate location in the path.This process then repeats until either the bundle is full or no bids can beat the current best bid proposed by other agents. A more formal description of this process is outlined below.

Computing the score for a task is a complex process which is dependent on the tasks already in the agent's path (and/or bundle). Selecting the best score for task j can be performed using the following two steps. First, task j is "inserted" in the path at some location n_j (the new path becomes $(\mathbf{p}_i \oplus_{n_j} j)$, where $\oplus_n$ signifies inserting the task at location n)[3]. The score for each task $c_j(\tau)$ is dependent on the time at which it is executed, motivating the second step, which consists of finding the optimal execution time given the new path, $\tau^\star_{ij}(\mathbf{p}_i \oplus_{n_j} j)$. This can be found by solving the following optimization problem:

$$\begin{aligned} \tau^\star_{ij}(\mathbf{p}_i \oplus_{n_j} j) = \quad & \underset{\tau_{ij}\in[0,\infty)}{\operatorname{argmax}}\, c_j(\tau_{ij}) \\ \text{subject to:} \quad & \tau^\star_{ik}(\mathbf{p}_i \oplus_{n_j} j) = \tau^\star_{ik}(\mathbf{p}_i), \quad \forall k \in \mathbf{p}_i. \end{aligned} \tag{3.6}$$

[3]The notion of inserting task j into the path at location n_j involves shifting all path elements from n_j onwards by one and changing path element at location n_j to be task j (i.e $p_{i(n+1)} = p_{in}, \forall n \geq n_j$ and $p_{in_j} = j$)

The constraints state that the insertion of the new task j into path $\mathbf{p}_i$ cannot impact the current times (and corresponding scores) for the tasks already in the path [51]. Note that this is a continuous time optimization, which, for the general case, involves a significant amount of computation. The optimal score associated with inserting the task at location n_j is then given by $c_j(\tau_{ij}^\star(\mathbf{p}_i \oplus_{n_j} j))$. This process is repeated for all n_j by inserting task j at every possible location in the path. The optimal location is then given by,

$$n_j{}^\star = \underset{n_j}{\operatorname{argmax}}\, c_j(\tau_{ij}^\star(\mathbf{p}_i \oplus_{n_j} j)) \tag{3.7}$$

and the final score for task j is $c_{ij}(\mathbf{p}_i) = c_j(\tau_{ij}^\star(\mathbf{p}_i \oplus_{n_j{}^\star} j))$.

Once the scores for all possible tasks are computed ($c_{ij}(\mathbf{p}_i)$ for all $j \notin \mathbf{p}_i$)), the scores need to be checked against the winning bid list, $\mathbf{y}_i$, to see if any other agent has a higher bid for the task. We define the variable $h_{ij} = \mathbb{I}(c_{ij}(\mathbf{p}_i) > y_{ij})$, where $\mathbb{I}(\cdot)$ denotes the indicator function that equals unity if the argument is true and zero if it is false, so that $c_{ij}(\mathbf{p}_i)h_{ij}$ will be nonzero only for viable bids. The final step is to select the highest scoring task to add to the bundle:

$$j^\star = \underset{j \notin \mathbf{p}_i}{\operatorname{argmax}}\, c_{ij}(\mathbf{p}_i)h_{ij} \tag{3.8}$$

The bundle, path, times, winning agents and winning bids vectors are then updated to include the new task:

$$\begin{aligned}
\mathbf{b}_i &\leftarrow (\mathbf{b}_i \oplus_{\text{end}} j^\star) \\
\mathbf{p}_i &\leftarrow (\mathbf{p}_i \oplus_{n_j{}^\star} j^\star) \\
\boldsymbol{\tau}_i &\leftarrow (\boldsymbol{\tau}_i \oplus_{n_j{}^\star} \tau_{ij^\star}^\star(\mathbf{p}_i \oplus_{n_j{}^\star} j^\star)) \\
z_{ij} &= i \\
y_{ij} &= c_{ij^\star}(\mathbf{p}_i)
\end{aligned} \tag{3.9}$$

The bundle building recursion continues until either the bundle is full (the limit L_t is

reached), or no tasks can be added for which the agent is not outbid by some other agent ($h_{ij} = 0$ for all $j \notin \mathbf{p}_i$). Notice that with Equation (4.4), a path is uniquely defined for a given bundle, while multiple bundles might result in the same path.

Phase 2: Consensus

Once agents created their bundles, they need to communicate with all of the other tasks in order to resolve conflicting assignments amongst the team. This communication takes the form of each agent broadcasting its current values for $\mathbf{z}_i$,$\mathbf{y}_i$ and $\mathbf{s}_i$. After each agent receives this information from its neighboring agents, each agent can determine if it has been outbid for any task in its bundle. Since the bundle building recursion, described in the previous section, depends at each iteration upon the tasks in the bundle up to that point, if an agent is outbid for a task, it must release it and all subsequent tasks from its bundle. If the subsequent tasks are not released, then the current best scores computed for those tasks would be overly conservative, possibly leading to a degradation in performance. It is better, therefore, to release all tasks after the outbid task and redo the bundle building recursion process to re-add these tasks with more accurate bids (or possibly better ones) back into the bundle.

This consensus phase assumes that each pair of neighboring agents *synchronously* shares the following information vectors: the winning agent list $\mathbf{z}_i$, the winning bids list $\mathbf{y}_i$, and the vector of timestamps $\mathbf{s}_i$ representing the time stamps of the last information updates received about all the other agents. The timestamp vector for any agent i is updated using the following equation,

$$s_{ik} = \begin{cases} \tau_r, & \text{if } g_{ik} = 1 \\ \max\{s_{mk} \mid m \in \mathcal{I}, g_{im} = 1\} & \text{otherwise,} \end{cases} \tag{3.10}$$

which states that the timestamp s_{ik} that agent i has about agent k is equal to the message reception time τ_r if there is a direct link between agents i and k (i.e. $g_{ik} = 1$ in the network graph), and is otherwise determined by taking the latest timestamp about agent k from the set of agent i's neighboring agents.

For each message that is passed between a sender k and a receiver i, a set of actions is executed by agent i to update its information vectors using the received information. These actions involve comparing its vectors $\mathbf{z}_i$, $\mathbf{y}_i$, and $\mathbf{s}_i$ to those of agent k to determine which agent's information is the most up-to-date for each task. There are three possible actions that agent i can take for each task j:

1. **Update**: $z_{ij} = z_{kj},\ \ y_{ij} = y_{kj}$
2. **Reset**: $z_{ij} = \emptyset,\ \ y_{ij} = 0$
3. **Leave**: $z_{ij} = z_{ij},\ \ y_{ij} = y_{ij}$.

The decision rules for this synchronous communication protocol were originally presented in [32] and are provided below in in Table 3.1 . The first two columns of the table indicate the agent that each of the sender k and receiver i believes to be the current winner for a given task; the third column indicates the action that the receiver should take, where the default action is "Leave". In Section 4.1 we present a revised communication protocol to handle *asynchronous* communication.

If either of the winning agent or winning bid information vectors ($\mathbf{z}_i$ or $\mathbf{y}_i$) are changed as an outcome of the communication, the agent must check if any of the updated or reset tasks were in its bundle. If so, those tasks, along with all others added to the bundle after them, are released. Thus if $\bar{n}$ is the location of the first outbid task in the bundle ($\bar{n} = \min\{n \mid z_{i(b_{in})} \neq i\}$ with b_{in} denoting the n^{th} entry of the bundle), then for all bundle locations $n \geq \bar{n}$, with corresponding task indices b_{in}, the following updates are made:

$$\begin{aligned} z_{i(b_{in})} &= \emptyset \\ y_{i(b_{in})} &= 0, \end{aligned} \tag{3.11}$$

The bundle is then truncated to remove these tasks,

$$\mathbf{b}_i \leftarrow \{b_{i1}, \ldots, b_{i(\bar{n}-1)}\} \tag{3.12}$$

Table 3.1: CBBA Action rules for agent i based on communication with agent k regarding task j

Agent k (sender) thinks z_{kj} is	**Agent i (receiver) thinks z_{ij} is**	**Receiver's Action (default: leave)**
k	i	if $y_{kj} > y_{ij} \rightarrow$ update
	k	update
	$m \notin \{i, k\}$	if $s_{km} > s_{im}$ or $y_{kj} > y_{ij} \rightarrow$ update
	none	update
i	i	leave
	k	reset
	$m \notin \{i, k\}$	if $s_{km} > s_{im} \rightarrow$ reset
	none	leave
$m \notin \{i, k\}$	i	if $s_{km} > s_{im}$ and $y_{kj} > y_{ij} \rightarrow$ update
	k	if $s_{km} > s_{im} \rightarrow$ update else $\rightarrow$ reset
	m	$s_{km} > s_{im} \rightarrow$ update
	$n \notin \{i, k, m\}$	if $s_{km} > s_{im}$ and $s_{kn} > s_{in} \rightarrow$ update if $s_{km} > s_{im}$ and $y_{kj} > y_{ij} \rightarrow$ update if $s_{kn} > s_{in}$ and $s_{im} > s_{km} \rightarrow$ reset
	none	if $s_{km} > s_{im} \rightarrow$ update
none	i	leave
	k	update
	$m \notin \{i, k\}$	if $s_{km} > s_{im} \rightarrow$ update
	none	leave

and the corresponding entries are removed from the path and times vectors as well. From here, the algorithm returns to the first phase where new tasks can be added to the bundle. CBBA iterates between these two phases until no changes to the information vectors occur anymore.

Scoring Functions

It has previously been shown that if the scoring function satisfies a certain condition, called *diminishing marginal gain* (DMG), CBBA is guaranteed to produce a conflict-

free assignment and converge in at most $\max\{N_t, L_t N_a\} D$ iterations, where D is the network diameter (always less than N_a) [32]. The DMG property states that the score for a task cannot increase as other elements are added to the set before it. In other words,

$$c_{ij}(\mathbf{p}_i) \geq c_{ij}(\mathbf{p}_i \oplus_n m) \tag{3.13}$$

for all $\mathbf{p}_i$, n, m, and j, where $m \neq j$ and $m, j \notin \mathbf{p}_i$.

Many reward functions in search and exploration problems for UAVs satisfy the DMG condition. The present authors have shown in [32, 51] that DMG is satisfied for the following two cases: (a) time-discounted rewards, and (b) more generally time-windowed rewards.

Time-Discounted Reward Consider the following time-discounted reward [12, 52, 53] that has been commonly used for UAV task allocation problems:

$$c_{ij}(\mathbf{p}_i) = \lambda_j^{\tau_{ij}(\mathbf{p}_i)} R_j \tag{3.14}$$

where $\lambda_j < 1$ is the discount factor for task j, $\tau_{ij}(\mathbf{p}_i)$ is the estimated time agent i will take to arrive at task location j by following path $\mathbf{p}_i$, and R_j is a static reward associated with performing task j. The time-discounted reward can model search scenarios in which uncertainty growth with time causes degradation of the expected reward for visiting a certain location. Equation 3.14 could also be used to model planning of service routes in which client satisfaction diminishes with time. Since the triangular inequality holds for the actual distance between task locations,

$$\tau_{ij}(\mathbf{p}_i \oplus_n m) \geq \tau_{ij}(\mathbf{p}_i) \tag{3.15}$$

for all n and all $m \neq j$, $m \notin \mathbf{p}_i$. In other words, if an agent moves along a longer path, it arrives at each of the task locations at a later time than if it had moved along a shorter path, resulting in a further discounted score value. Therefore, assuming the task rewards R_j are nonnegative for all j, the score function in Equation 3.14 satisfies DMG.

Time-Windowed Reward To incorporating scoring functions with more complicated temporal dependencies we break the score function into two parts:

1. *Time Window, $w_j(\tau)$:* The time window of validity for a task represents the time in which the task is allowed to be started. For task j this window is defined as

$$w_j(\tau) = \begin{cases} 1, & \tau_{j_{start}} \leq \tau \leq \tau_{j_{end}} \\ 0, & \text{otherwise} \end{cases} \tag{3.16}$$

2. *Score Profile, $s_j(\tau)$:* The score profile $s_j(\tau)$ represents the reward an agent receives from task j when it arrives at the task at time τ. This score is based on the reward for the task, R_j. For example, for the time-discounted case described above this quantity is $s_j(\tau) = \lambda_j^{\Delta\tau} R_j$, where $\Delta\tau = \max\{0, \tau - \tau_{j_{start}}\}$ is the difference between the task start time and the agent arrival time, and $\lambda_j < 1$ is the discount factor to penalize late arrivals. Without time discounting $s_j(\tau) = R_j$.

The score an agent receives for a task is a function of his arrival time at the task location, τ_{ij}, and can be computed as $c_j(\tau_{ij}) = s_j(\tau_{ij}) w_j(\tau_{ij})$. The arrival time, τ_{ij}, is in turn a function of the path the agent has taken before reaching task j, as described in the previous sections, and can be optimized as in Equation 4.1. Using time windows for tasks provides a framework to penalize early arrivals as well as late arrivals and accelerates the computation of the optimal task execution time by restricting the range of values that the arrival time can take.

To verify that the time-windows framework satisfies the DMG property we want to ensure that for all $j \notin \mathbf{p}_i$,

$$c_{ij}(\mathbf{p}_i) \geq c_{ij}(\mathbf{p}'_i),$$

where $\mathbf{p}'_i = \{\mathbf{p}_i \oplus_{n^\star_m} m\}$ such that $n^\star_m$ is the optimal location in the path for task $m \notin \mathbf{p}_i$, $m \neq j$, with a corresponding optimal time of $\tau^\star_{im}$. Note that the constraint set $\tau^\star_{ik}(\mathbf{p}'_i) = \tau^\star_{ik}(\mathbf{p}_i), \forall k \in \mathbf{p}_i$ is assumed to be satisfied during the addition of task m as described in Equation 4.1. For a new task $j \notin \mathbf{p}'_i$, the problem of computing $\tau^\star_{ij}$

given the new path $\mathbf{p}_i'$ becomes,

$$\begin{aligned} \tau_{ij}^{\star}(\mathbf{p}_i' \oplus_{n_j} j) = \quad & \underset{\tau_{ij} \in [0,\infty)}{\operatorname{argmax}} c_j(\tau_{ij}) \\ \text{subject to:} \quad & \tau_{ik}^{\star}(\mathbf{p}_i' \oplus_{n_j} j) = \tau_{ik}^{\star}(\mathbf{p}_i'), \quad \forall k \in \mathbf{p}_i'. \end{aligned} \tag{3.17}$$

for each insertion location n_j. The constraint set for this optimization can be rewritten as the following set of constraints,

$$\tau_{ik}((\mathbf{p}_i \oplus_{n_m^{\star}} m) \oplus_{n_j} j) = \tau_{ik}^{\star}(\mathbf{p}_i \oplus_{n_m^{\star}} m) = \tau_{ik}^{\star}(\mathbf{p}_i), \quad \forall k \in \mathbf{p}_i \tag{3.18}$$

$$\tau_{im}((\mathbf{p}_i \oplus_{n_m^{\star}} m) \oplus_{n_j} j) = \tau_{im}^{\star}(\mathbf{p}_i \oplus_{n_m^{\star}} m) \tag{3.19}$$

Note that the second equality represents an additional constraint to the original set corresponding to inserting task j in path $\mathbf{p}_i$. In fact, with each new task that is inserted into the path one additional constraint must be satisfied. Since at each iteration of the bundle building process we are solving a more constrained problem than we would have with a shorter bundle, the optimal score for tasks can only decrease, thus satisfying DMG.

3.5 CBBA Analysis

The following section will give some insight into the limitations of the CBBA algorithm in the environments considered in this thesis.

3.5.1 Synchronous Nature of CBBA

CBBA was originally designed to be decentralized. During the design process, an effort was made to allow agents as much autonomy in creating bids as possible but the final algorithm was tightly synchronized. During experimental testing it became obvious that there are many logistical challenges to implementing synchronization in decentralized algorithms. This process involved creating many candidate solutions for implementing this decentralized synchronization. Figure 3-1 highlights the three

main challenges associated with this decentralized synchronization.

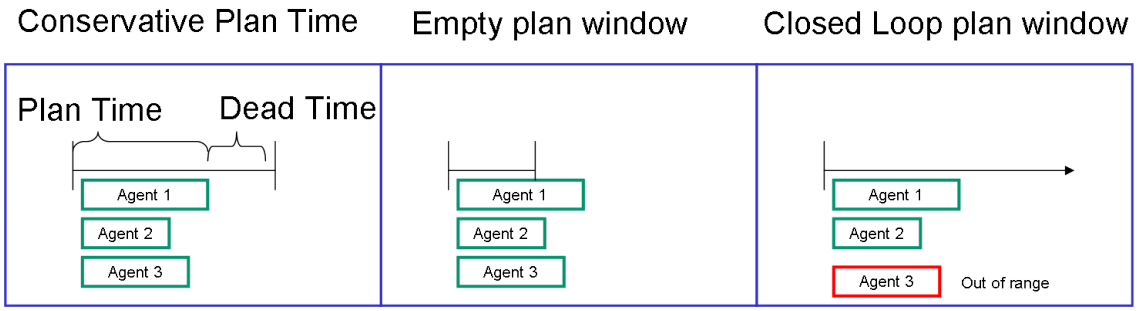


Figure 3-1: Three difficulties with decentralized synchronization. The width of the boxes (labeled as agents 1-3) represents the computation time for the bundle building phase. The black bar above these agents represents the globally set heartbeat time.

One set of candidate solutions for synchronization in a decentralized environment is the idea of a global heartbeat. In this environment, all of the decentralized agents would observe this global heartbeat to synchronize every consensus step in CBBA. As can be seen in the "conservative plan time" block, if the heartbeat is set for too long, then computation time is wasted by forcing all agents that have finished early to idle. On the other hand, as can be seen in the "empty plan window block", if the heartbeat time is set too short, then agents may not have finished their computation in time to join every iteration. In this situation, the agents create what is effectively an artificial dynamic network, where only a subset of the agents communicate at every iteration. When implemented in practice, both the problems of "conservative plan time" and "empty plan windows" are seen at every single iteration for different sets of agents. Another solution was introduced in which all agents wait for all other agents to return with assignments, then only advance once they had heard from everyone. This fixed the problem with creating an artificial dynamic network, but introduced a severe problem if the network *actually* was dynamic. In that case the algorithm would never converge because agents would be "stuck" waiting for a message that was not going to arrive. It was unclear how to scale this method to handle dynamic networks with possible dropped connectivity in a fast reliable way. It turned out that these synchronous effects were crippling the convergence of the algorithm when communication was anything other than perfect. Only in a very

reliable communication environments would CBBA perform as had been expected. Given these observations, it was clear that robust decentralized algorithms couldn't heavily rely on global synchronization.

3.5.2 Information Incorporated During the Plan Cycle

One important consideration was to define where in the plan cycle new information was allowed to be introduced. This choice is a direct trade-off between faster response times to changes in situational awareness and convergence stability. Traditionally, real-time planners utilize the notion of a replan to deal with changes in the situational awareness of the fleet. This consisted of creating some global heartbeat where the fleet completely rebuilt its task allocation at every one of these replan iterations. There are two implicit assumptions with this methodology: 1) The time that it takes the planner to replan is relatively short; and 2) the replan iterations can be scheduled frequently enough such that there are no significant delays in situational awareness propagation into the plan. In decentralized architectures assumption 1 is usually a non-negligible time. In addition, assumption 2 that replans can be run arbitrarily often is broken when assumption 1 breaks because the new replan will be throttled by how long it takes for the first plan to converge (and more importantly how long it takes the fleet to realize that it has converged). It becomes clear in real-time environments that the ability to include information mid-plan cycle is extremely beneficial for adaptation to changes in situational awareness as long as this effort does not significantly affect convergence. CBBA was burdened by the constraint of global replans, and long decentralized convergence rates. This observation made it clear that robust decentralized algorithms needed much faster tools for incorporating changes in situational awareness into plans. A good way to accomplish this seemed to allow more local replan decisions over the global replans that CBBA requires.

3.5.3 Message Passing

The ways that messages are bundled and propagated defines the complexity of the consensus protocol needed to interpret the information. The simplest approach is to just forward every single packet so that information is always propagating directly from the source. This forms artificially strongly-connected networks with the side effect of propagating redundant, and at times, incorrect information around the network. To counter this effect, planners like CBBA use a consensus protocol. It allows each agent to look at the information it is receiving from other agents in the network and combine it into self-consistent bundles of information, without having to repeat old information or forward information that it already knows to be incorrect. This consensus process does, at times, also lead to tricky problems where information can confuse the consensus algorithm. This forces these consensus algorithms to be overly conservative and require more total information to enter the network (but often significantly fewer actual messages) than would have been needed to propagate every message to all agents.

3.5.4 Convergence Definition

The definition of convergence has a very natural meaning when we are talking about global convergence in a static environment, but this notion breaks down in decentralized or dynamic environments. This has lead to distinctions in how to quantify and correctly identify convergence. The most robust way to handle convergence in static environments is to create a convergence "handshake" so every agent is able to confirm that they have converged with every other agent. This solution becomes more difficult in dynamic networks because agents are entering and leaving the network, making it difficult to tell the difference between communication drops and just long series of computations. Thus, on a global setting, convergence becomes very difficult to verify in decentralized applications. In fact it will be shown in Chapter 4 that in some cases (very large, or highly dynamic environments), global convergence does not even ever exist.

This observation that there may not even exist a notion of global convergence motivates the need for some type of local convergence. Local convergence can be defined as recognizing when all local computations and decisions have come to their final values. However, even local convergence suffers from additional complications. When agents are interacting with each other, local convergence may often only be temporary. Local convergence metrics are much less robust at identifying algorithm completion than global convergence but most of the time they allow the agents to make productive decisions while they are waiting for actual global convergence (if global convergence even exists for the particular environment). This lead to another realization that any truly decentralized planner would have to recognize some notion of local convergence to operate in dynamic environments.

3.5.5 When to Start Executing a Plan?

When we start dealing with local convergence instead of global convergence (because global convergence is either hard to detect or never occurs) a decision needs to made about when a stable plan has been generated and can safely start being executed. This is different from determining local or even global convergence because this decision only focuses on minimizing wasted effort by the agent. This means that even if a particular agent is going to change its world belief, as long as its plan (or just the first item in its plan) doesn't change it can safely execute it. This is a very difficult question to robustly answer before a global convergent state. If reliable methods to determine the answer to this question are obtained, the notion of convergence becomes less central. In our ACBBA work that will be described in Chapter 4, we have determined a parameter, called *separation time*, that does a reasonably good job of estimating when a plan is executable, but more robust ways are still needed. If we could create a process of determining when to start executing plans based on bid logic or using other information in bids that is currently discarded, it would be tremendously beneficial for our decentralized planning approaches.

3.5.6 Algorithmic Implications

The five considerations listed above translate almost directly into a set of features that a decentralized task planning algorithm with our domain constraints will need to have. These are:

1. There cannot be a strong notion of global synchronization needed for algorithmic convergence

2. There should be a way for local changes in situational awareness to enter the plan cycle without requiring a global replan

3. Messages should be handled as efficiently as possible, ideally not requiring any global synchronization

4. There need to be some local measures to detect convergence in dynamic environments

5. Start servicing tasks as soon as it's clear what the next best thing to do is, as opposed to waiting for some measure of convergence.

These features were the key motivators for transitioning from CBBA to the asynchronous consensus based bundle algorithm (ACBBA) presented in Chapter 4.

Chapter 4

The Asynchronous Consensus Based Bundle Algorithm

Originally the consensus based bundle algorithm (CBBA), introduced in the previous chapter, was designed as a decentralized task allocation algorithm. Unfortunately in environments where communications were not ideal, the implicit synchronous assumptions impeded the the fluid execution and the algorithm didn't perform as expected. The following key features were identified that would be necessary to transition CBBA into an algorithm that could be used in decentralized environments:

1. Depart from all of the implicit "global" notions in algorithmic operations
2. Use available information as efficiently as possible.
3. Allow agent consensus and convergence to be determined independently and locally.
4. Allow agents to enter and leave the network and allow tasks to be created and deleted in real time.

In the spirit of fixing these problems while leveraging the decisions and insights that were introduced in Chapter the Asynchronous Consensus Based Bundle Algorithm (ACBBA) was created.

4.1 Asynchronous CBBA (ACBBA)

4.1.1 Algorithmic Discussion

ACBBA retains the 2 phase structure that was introduced with both the sequential greedy algorithm and CBBA. The main differences come in the synchronization of the phases, and the assumptions that need to be made in the consensus protocol. The theme introduced with ACBBA was to enable the agents to operate independently of all other agents in the fleet. In the spirit of the goals of Chapter , the agents would do their best to come to a conflict free assignment on the assignment matrix $\mathbf{x}$ but the algorithm would not fail when communication and global structures were not available. Because of this, each agent was allowed to build bundles and perform consensus on their own schedule. This was made possible through a careful understanding understanding of how to communicate with other agents, and with what information should be present in their communications. For completeness we will introduce a full description of the algorithm. Much of it will be similar to CBBA but the differences will be effective ways to address the goals that were identified at the beginning of this chapter.

Overall Algorithmic Flow

In order to create a robust algorithmic flow, the information passing inside of ACBBA had to be carefully controlled. The specific approach was implemented using two separate modules of the algorithm each running in parallel, a consensus module and a bundle building module. The consensus module is where message receiving and information deconfliction occurs. The bundle building module then takes this information, builds a local bundle, and rebroadcasts the results. An example of the overall system architecture is depicted in Figure 4-1.

The flow of information in ACBBA can be seen to start in the consensus phase when information is received from other agents or supervising users. This is the primary channel of receiving information about the outside world. All messages including information about other agent's desired allocations, to information about the

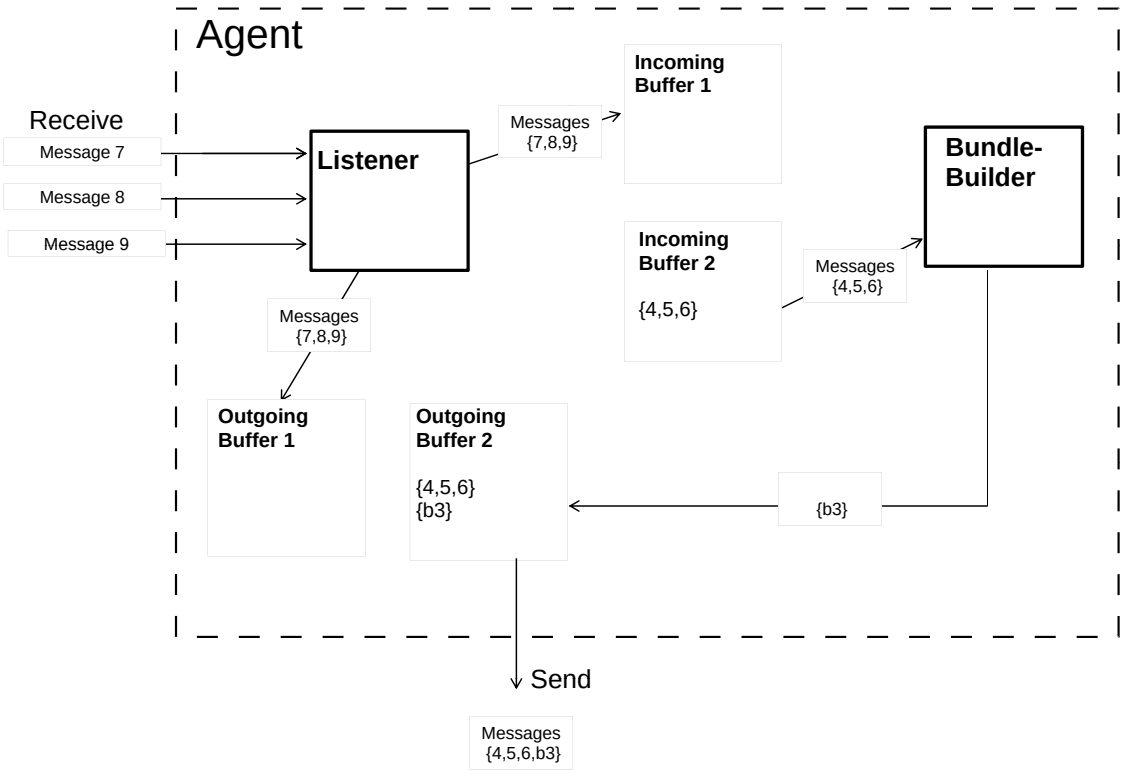


Figure 4-1: Agent decision architecture: Each agent runs two main threads, Listener and BundleBuilder; these two threads interface via the Incoming Buffers – there are two buffers to avoid unexpected message overriding; outgoing buffers manage rebroadcasting of information, triggered by the completion of the BundleBuilder thread.

creation or completion of other tasks enters through this channel. When information enters the consensus module that is relevant to algorithmic execution it is copied and sent to two locations: 1) the bundle building buffer, and 2) the rebroadcast buffer. The reason for the complexity added with these buffers is that managing the state inside the algorithm is very important to ensuring proper execution of the algorithm. This consensus module is run continuously so the algorithm is able to immediately assess the relevance of incoming information. To recap, relevant information enters through the consensus module, then is sent to both a bundle building buffer and a rebroadcast buffer.

We will discuss what happens to the information in the bundle building buffer first. This part of the algorithm is where we retain our notion of an "iteration."

Since computing new bundles is only worthwhile when enough relevant information is available, the execution of this module is throttled. This throttling is a local decision that can take the form of a timed iteration, or a closed loop process based on messages arriving in the bundle building buffer. In practice, a combination of the two is implemented. The bundle building process is spawned on an iteration time but with the caveat that it doesn't run unless new information is available. When the bundle building module is scheduled to run, the module attempts to build its bundle identically to CBBA. It reads in bids on tasks that the listener has said that it was outbid on, then repopulates its bundle with new tasks if necessary. A reminder of exactly what this process entails will be added in the next sections below. The result of this process will be a final bundle of tasks for this agent. Any changes to this agent's bundle are noted and sent to the same rebroadcast buffer that was holding the information that the agent just acted upon. At this point the rebroadcast function is called, all all relevant information is broadcast out to the agent's neighbors. In this rebroadcast buffer, there is usually a mixture of data from the consensus module and from the bundle building module. The information in the rebroadcast buffer that was sent from the consensus module is exactly the same information that the bundle building module received before it built its most recent bundle. Packing the rebroadcast information in this way allows updates made in the bundle building module to prune outdated information in the outgoing queue. This guarantees that the outgoing messages accurately represent the state of the current bundle build, saving computation resources and message bandwidth at the expense of a longer rebroadcast period. By carefully controlling the information exchange inside of the ACBBA algorithm we are able to make assumptions about the state of information that is propagated and can utilize these assumptions in the consensus module of the algorithm. The next 2 subsections will outline exactly what is occurring in each of the main modules of this algorithm.

Module 1: Bundle Construction

The main form of this module is when it is triggered to run, it takes all of new information from the bundle building buffer and updates its local bundle. After each agent receives this information, the agent can determine if it has been outbid for any task in its bundle. Since the bundle building process, described in the previous section, depends at each iteration upon the tasks in the bundle up to that point, if an agent is outbid for a task, it must release it and all subsequent tasks from its bundle. If the subsequent tasks are not released, then the current best scores computed for those tasks would be overly conservative, possibly leading to a degradation in performance. It is better, therefore, to release all tasks after the outbid task and redo the bundle building recursion process to re-add these tasks with more accurate bids (or possibly better ones) back into the bundle. Each agent will attempt to rebuild its own bundle in a greedy fashion until it is incapable of adding any more tasks (either due to lack of doable tasks or reaching the bundle limit L_i). The algorithm progresses by looking through the set of available tasks (an available task is a task that is not completed and is not already in the agent's bundle), computes a score for each task then checks this score against a list of the the current winning bids. The bid is kept as a candidate next best bid if it is greater than the current best bid for that task. After this process has been completed for all of the available tasks, the agent selects the bid with the highest score and inserts that bid to the end of its bundle and its appropriate location in the path.This process then repeats until either the bundle is full or no bids can beat the current best bid proposed by other agents. A more formal description of computing the actual bids is outlined below.

Computing the score for a task is a complex process which is dependent on the tasks already in the agent's path (and/or bundle). Selecting the best score for task j can be performed using the following two steps. First, task j is "inserted" in the path at some location n_j (the new path becomes $(\mathbf{p}_i \oplus_{n_j}, j)$, where $\oplus_n$ signifies inserting the task at location n)[1]. The score for each task $c_{ij}(t)$ is dependent on the time at

[1]The notion of inserting task j into the path at location n_j involves shifting all path elements from n_j onwards by one and changing path element at location n_j to be task j (i.e $p_{i(n+1)} = p_{in}, \forall n \geq n_j$

which it is executed, motivating the second step, which consists of finding the optimal execution time given the new path, $\tau_{ij}^{\star}(\mathbf{p}_i \oplus_{n_j} j)$. This can be found by solving the following optimization problem:

$$\begin{aligned} \tau_{ij}^{\star}(\mathbf{p}_i \oplus_{n_j} j) = \quad & \underset{\tau_{ij} \in [0,\infty)}{\operatorname{argmax}} c_j(\tau_{ij}) \\ \text{subject to:} \quad & \tau_{ik}^{\star}(\mathbf{p}_i \oplus_{n_j} j) = \tau_{ik}^{\star}(\mathbf{p}_i), \quad \forall k \in \mathbf{p}_i. \end{aligned} \tag{4.1}$$

The constraints state that the insertion of the new task j into path $\mathbf{p}_i$ cannot impact the current times (and corresponding scores) for the tasks already in the path [51]. Note that this is a continuous time optimization, which, for the general case, involves a significant amount of computation. The optimal score associated with inserting the task at location n_j is then given by $c_j(\tau_{ij}^{\star}(\mathbf{p}_i \oplus_{n_j} j))$. This process is repeated for all n_j by inserting task j at every possible location in the path. The optimal location is then given by,

$$n_j{}^{\star} = \underset{n_j}{\operatorname{argmax}}\, c_j(\tau_{ij}^{\star}(\mathbf{p}_i \oplus_{n_j} j)) \tag{4.2}$$

and the final score for task j is $c_{ij}(\mathbf{p}_i) = c_j(\tau_{ij}^{\star}(\mathbf{p}_i \oplus_{n_j{}^{\star}} j))$.

Once the scores for all possible tasks are computed ($c_{ij}(\mathbf{p}_i)$ for all $j \notin \mathbf{p}_i$)), the scores need to be checked against the winning bid list, $\mathbf{y}_i$, to see if any other agent has a higher bid for the task. We define the variable $h_{ij} = \mathbb{I}(c_{ij}(\mathbf{p}_i) > y_{ij})$, where $\mathbb{I}(\cdot)$ denotes the indicator function that equals unity if the argument is true and zero if it is false, so that $c_{ij}(\mathbf{p}_i)h_{ij}$ will be non-zero only for viable bids. The final step is to select the highest scoring task to add to the bundle:

$$j^{\star} = \underset{j \notin \mathbf{p}_i}{\operatorname{argmax}}\, c_{ij}(\mathbf{p}_i)h_{ij} \tag{4.3}$$

At this point the bundle, path, winning agents and winning bids vectors are then

and $p_{in_j} = j$)

updated to include the new task:

$$\begin{aligned} \mathbf{b}_i &\leftarrow (\mathbf{b}_i \oplus_{\text{end}} j^\star) \\ \mathbf{p}_i &\leftarrow (\mathbf{p}_i \oplus_{n_{j^\star}} j^\star) \\ \boldsymbol{\tau}_i &\leftarrow (\boldsymbol{\tau}_i \oplus_{n_{j^\star}} \tau^\star_{ij^\star}(\mathbf{p}_i \oplus_{n_{j^\star}} j^\star)) \\ z_{ij} &= i \\ y_{ij} &= c_{ij^\star}(\mathbf{p}_i) \end{aligned} \tag{4.4}$$

In addition to this, the current time is sampled and recorded as the bid time for this bid. This bid time is important during the consensus phase.

The bundle building recursion continues until either the bundle is full (the limit L_t is reached), or no tasks can be added for which the agent is not outbid by some other agent ($h_{ij} = 0$ for all $j \notin \mathbf{p}_i$). Notice that with Equation (4.4), a path is uniquely defined for a given bundle, while multiple bundles might result in the same path.

Phase 2: Consensus

Once agents created their bundles, they need to communicate with all of the other tasks in order to resolve conflicting assignments amongst the team. This module is responsible for interpreting the information arriving form the agents team members. Recall that during CBBA consensus local variables on each agent. These variables for each agent i were the winning agent list $\mathbf{z}_i$ and the winning bid list $\mathbf{y}_i$. Previously there was a variable for decision timestamps $\mathbf{s}_i$ that specified the time in which 2 agents communicated with each other. This worked fine in synchronized settings but as we moved into more asynchronous settings we needed to be clearer about what the time stamp represented. With ACBBA we created a similar variable $\mathbf{t}_i$ that represents the time that the bid was relevant. This was necessary because what is really important about this time is not how recently you heard the information but how relevant the information is. Changing this time variable allows us to create a much more intelligent task consensus protocol that directly targets the sources of conflicted information.

Another useful result of clarifying our understanding of the propagation of the information state, is that we are able to empower the consensus algorithm to control when information should be propagated. In the terminology of this protocol, we call this information propagation a rebroadcast. What this rebroadcast decision ensures is that each agent then only broadcasts new or relevant information, effectively reducing the overall bandwidth requirements of a decentralized system. In order to to create a robust task consensus protocol, ACBBA always assumes worst case networking scenarios. This robustness is what is able to handle dynamic network topologies naturally, allowing agents to enter and leave the network as well as links to drop and reappear. This new protocol ends up working well in decentralized environments because the task consensus process is run using only local information, therefore even in worst case network scenarios, informed decisions can be made. Below is an outline of the message consensus rules. They are fairly complicated, but exhaustive to capture all possible scenarios and enable convergence.

The above table describes what actions agent i should take after receiving a message from agent k. The term z_{kj} refers to agent k's belief of who won task j and y_{kj} represents the associated winning bid. The new term t_{kj} refers to the timestamp of when this winning bid was made.

4.1.2 Local Deconfliction Rules of ACBBA

1. **Update & Rebroadcast:** The receiver i updates its winning agent z_{ij}, winning bid y_{ij}, and winning time t_{ij} with the received information from the sender k. It then propagates this new information.

2. **Leave & Rebroadcast:** The receiver does not change its information state, but rebroadcasts its local copy of the winning agent's information because either it believes its information is more correct than the sender's, or the agent is unsure and it's looking for confirmation from another agent.

3. **Leave & No-Rebroadcast:** The receiver neither changes its information state nor rebroadcasts it. This action is applied when the information is either

Table 4.1: ACBBA decision rules for agent i based on communication with agent k regarding task j (z_{uj}: winning agent for task j from agent u's perspective; y_{uj}: winning bid on task j from agent u's perspective; t_{uj}: timestamp of the message agent u received associated with the current z_{uj} and y_{uj})

	Agent k (sender) thinks z_{kj} is	**Agent i (receiver) thinks z_{ij} is**	**Receiver's Action (default: leave & rebroadcast)**
1	k	i	if $y_{kj} > y_{ij} \rightarrow$ update† & rebroadcast
2			if $y_{kj} = y_{ij}$ and $z_{kj} < z_{ij} \rightarrow$ update & rebroadcast
3			if $y_{kj} < y_{ij} \rightarrow$ update‡ & rebroadcast
4		k	if $t_{kj} > t_{ij} \rightarrow$ update & rebroadcast
5			$\lvert t_{kj} - t_{ij} \rvert < \epsilon_t \rightarrow$ leave & no-broadcast
6			if $t_{kj} < t_{ij} \rightarrow$ leave & no-rebroadcast
7		$m \notin \{i, k\}$	if $y_{kj} > y_{ij}$ and $t_{kj} \geq t_{ij} \rightarrow$ update & rebroadcast
8			if $y_{kj} < y_{ij}$ and $t_{kj} \leq t_{ij} \rightarrow$ leave & rebroadcast
9			if $y_{kj} = y_{ij} \rightarrow$ leave & rebroadcast
10			if $y_{kj} < y_{ij}$ and $t_{kj} > t_{ij} \rightarrow$ update & rebroadcast
11			if $y_{kj} > y_{ij}$ and $t_{kj} < t_{ij} \rightarrow$ update & rebroadcast
12		none	update & rebroadcast
13	i	i	if $\lvert t_{kj} - t_{ij} \rvert < \epsilon_t \rightarrow$ leave‡ & no-rebroadcast
14		k	reset & rebroadcast*
15		$m \notin \{i, k\}$	leave‡ & rebroadcast
16		none	leave & rebroadcast*
17	$m \notin \{i, k\}$	i	if $y_{kj} > y_{ij} \rightarrow$ update† & rebroadcast
18			if $y_{kj} = y_{ij}$ and $z_{kj} < z_{ij} \rightarrow$ update & rebroadcast
19			if $y_{kj} < y_{ij} \rightarrow$ update‡ & rebroadcast
20		k	update & rebroadcast
21		m	if $t_{kj} > t_{ij} \rightarrow$ update & rebroadcast
22			$\lvert t_{kj} - t_{ij} \rvert < \epsilon_t \rightarrow$ leave & no-rebroadcast
23			if $t_{kj} < t_{ij} \rightarrow$ leave & rebroadcast
24		$n \notin \{i, k, m\}$	if $y_{kj} > y_{ij}$ and $t_{kj} \geq t_{ij} \rightarrow$ update & rebroadcast
25			if $y_{kj} < y_{ij}$ and $t_{kj} \leq t_{ij} \rightarrow$ leave & rebroadcast
26			if $y_{kj} < y_{ij}$ and $t_{kj} > t_{ij} \rightarrow$ update & rebroadcast
27			if $y_{kj} > y_{ij}$ and $t_{kj} < t_{ij} \rightarrow$ leave & rebroadcast
28		none	update & rebroadcast
29	none	i	leave‡ & rebroadcast
30		k	leave‡ & rebroadcast
31		$m \notin \{i, k\}$	if $t_{kj} > t_{ij} \rightarrow$ update & rebroadcast
32		none	leave & no-rebroadcast
	NOTE:	rebroadcast	With "leave," broadcast own information With "update," broadcast sender's information With "reset," broadcast sender's information
		rebroadcast*	empty bid with current time

not new or is outdated and should have been corrected already.

4. **Reset & Rebroadcast:** The receiver resets its information state: $z_{ij} = \emptyset$ and $y_{ij} = 0$, and rebroadcasts the original *received* message so that the confusion can be resolved by other agents.

5. **Update Time & Rebroadcast:** This case happens when the receiver is the task winner and observes a possibly confusing message. The receiver updates the timestamp on his bid to reflect the current time, confirming that the bid is still active at the current time.

Remarks on Key Decision Rules

• **Both Sender and Receiver think they win (Lines 1-3):** When this case arises we choose the highest of the two bids and call it the winner. To help with the propagation of this information throughout the network we make sure that the winning bid has a later time than the losing bid, if it does we do not change the time, if it does not, we update the time to be ϵ greater than the losing time. This if the winning agent is updating the time, it is confirming that at that particular time, it is still winning the task. If the receiving agent is updating the time, it is confirming that it in fact as been outbid on the task after than the time of its last bid. If the two agents have the exact same score, the tie breaker we use is the lowest agent ID. Given the continuous nature of our cost function, this tie case is very unlikely in any real situation.

• **Both Sender and Receiver think the sender won (Lines(4-6))** This is one of the convergent states because both agents agree on the winner. There is a small update made to propagate the most recent time so that both agents are always using the most recent time. If the times also match then this is a sink state where no extra messages are sent out.

• **Sender thinks that he won, while receiver thinks that some other agent not a sender or receiver has won (7-11)** This is one of the main confusion

resolving states with several rules. The easiest rule is that if the score and time are both greater for the sender then trust this information. Likewise, if the score and the time are less than the receiver then trust the receivers information. In the degenerate case when the scores are the same we also trust the lowest score. If there is confusion where the winning score and the times do not define a clear winner then we enter an unstable state where the agent always trusts the later timestamp. This case, the agent knows this will not be the final information state for this information, but it propagates the most recent time information to ask for clarification. If the agent with a lower score with a later time is the actual winner then it will always hold on to this information, if the agent with a higher score but an earlier time is the actual winner then the agent will receive this bid again with an updated time when the appropriate agent hears the message.

• **Sender thinks he won, Receiver has no bid knowledge (12)** In this case the agent is receiving a new bid so it trusts this information.

• **Both Sender and Receiver think Receiver is the winner (13)** If the two agents have the same information then this is a sink state. If the receiver has old information, then update. If the sender has an earlier time (then this is an old message that has been lost) ignore it, and rebroadcast the newer time.

• **Sender thinks receiver is the winner, Receiver thinks Sender is winner (14)** This state is a confused state and only occurs when there has been lots of recent changes to the agents bundles for this particular task. This is another unstable state where both agents broadcast an empty bid with the current time to say that neither agent thinks it is currently winning the task.

• **Sender thinks Receiver w/o and Receiver thinks someone other than the 2 has won (15)** In this case we just leave and rebroadcast the current Receiver state. The idea behind this is that we are hoping that the message we just received is old information and it will sort itself out.

• **Sender thinks Receiver has won when the receiver has no winner (16))**

This is a case of old information entering the network. The receiver just broadcasts the empty bid, insuring that the time its broadcasting for, is later than the one posted by the sender.

• **Sender thinks someone else won, Receiver thinks receiver won (17-19)** In this case the receiver looks at the score of the bid, if the receiver is outbid he accepts the senders bid and updates the time to be later than his if necessary. In the degenerate case if both the agents are tied, take the lowest Id. If the receivers bid is higher than the other, then rebroadcast the senders bid with an updated time if its necessary to make it later, otherwise leave the time unchanged. We can do this because the receiver can guarantee the bid that he himself made, even at an earlier time, so this is a robust consensus decision.

• **Sender thinks someone else, receiver thinks sender (20)** In this case the receiver will always trust the sender if the sender's time is greater than the message time. If the receivers time is greater than we know that the message sent was an old message and its safe to ignore, we rebroadcast our current information to clarify other agents or allow the sender to actually update the time.

• **Sender and Receiver agree on winner but its neither of them (21-23)** Under this situation if a newer time is received from the sender then the local information is updated, otherwise this state is a sink state.

• **Neither Sender or Receiver have won task, but they disagree on who has (24-27)** The straightforward rules are if the score is more and the time is later then trust the information, and it if the score is less and the time is earlier then discard information and rebroadcast the local copy. The other 2 conditions handle the unstable states. These are states where the local information the receiver has will most likely change, so the rule to use is to just trust the most recent information (later time), and allow the rest of the protocol to sort out the problem.

• **Sender is propagating a bid to an empty Receiver (28)** In this case we have the receiver trust the later of the two timesteps. This is because it is easy for the bids

of dropped tasks to be floating around in the network. the only way to pull them out is to trust that timestamps will be updated efficiently and newer times will prevail.

• **Sender thinks none and Receiver thinks it won (29)** In this case we just rebroadcast our own bid while updating the time if necessary to propagate the information that the sender thinks that it has won.

• **Sender thinks no one, receiver thinks Sender (30)** This is the propagation step for a dropped bid. This is important to start the propagation of dropped bids.

• **Sender thinks none, receiver thinks someone else (31)** Since its difficult to tell who has most accurate information, best we can do is trust later time and propagate that information.

• **Both think none** If both messages have the same time, then this is a sink state, if they have different times, trust the later time and rebroadcast.

Scoring Functions

For brevity here we will refer you to the scoring discussion in the previous chapter because it is identical to the one used with ACBBA.

4.1.3 Algorithmic Convergence

As a feature of the algorithm, ACBBA runs the Bundle Building and Consensus Phases simultaneously and continuously in separate threads. This added flexibility changes the way that we must think of convergence. In this continuously running system there is never a point in which CBBA ceases to execute. This means that other criteria need to be constructed to recognize that the system has entered a stable, conflict-free assignment. Furthermore, this criterion is also required to be a local measure, such that each agent can decide individually.

In a static environment ACBBA is guaranteed to converge. The convergence problem shown in Figure 4-2 illustrates how we think about full network convergence in the ACBBA algorithm. The scenario in this figure is for 8 agents, with a maximum

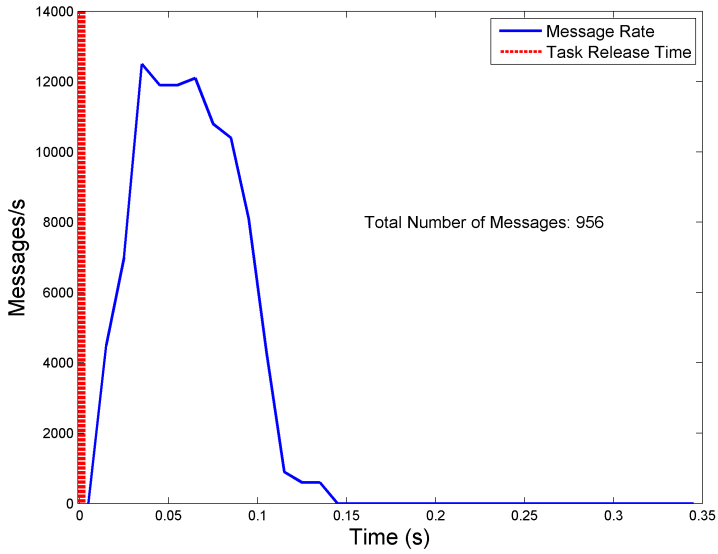
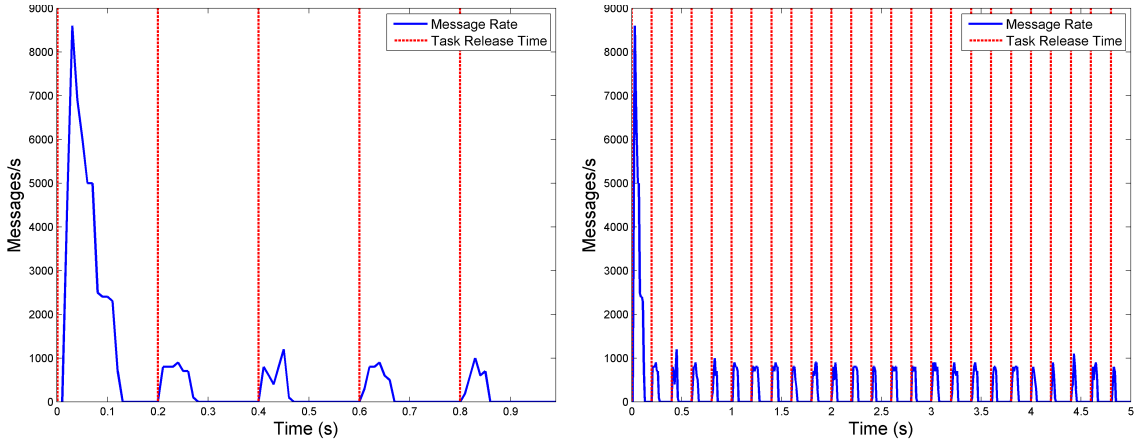


Figure 4-2: Message profile for a convergence iteration where all the tasks are released at the start of the program

bundle size of 5, bidding on 35 tasks. As we see in the figure, there is a large spike within the first 10 ms of the tasks being released. This heavy message passing regime continues for about 0.1 s, then, after some clean up messages, settles down to 0 messages being sent. Once the network goes quiet in this configuration we say that the network has converged. For the rest of the results in this paper, we define 1 message being sent as an agent broadcasting some piece of information to every vehicle that can hear it, no matter how many vehicles hear this one message being broadcast, we count it as 1 message. Also, for reference, throughout this entire process 956 messages were sent.

The case shown in Figure 4-2 is interesting, but it ignores one of the big regimes in which ACBBA was designed to operate. In Figure 4-3 we show an example of what the network looks like under the presence of pop-up tasks. The parameters for this test are identical the above scenario, except, instead of releasing all 35 tasks at once, an initial allocation of 10 tasks starts off the ACBBA convergence, then every 0.2 seconds after, a single pop-up task is added. In this scenario we can see that the initial convergence takes a similar form to what was shown in Figure 4-2. At 0.2 seconds we see that the first pop-up task is released. The algorithm has a small



(a) Early Convergence

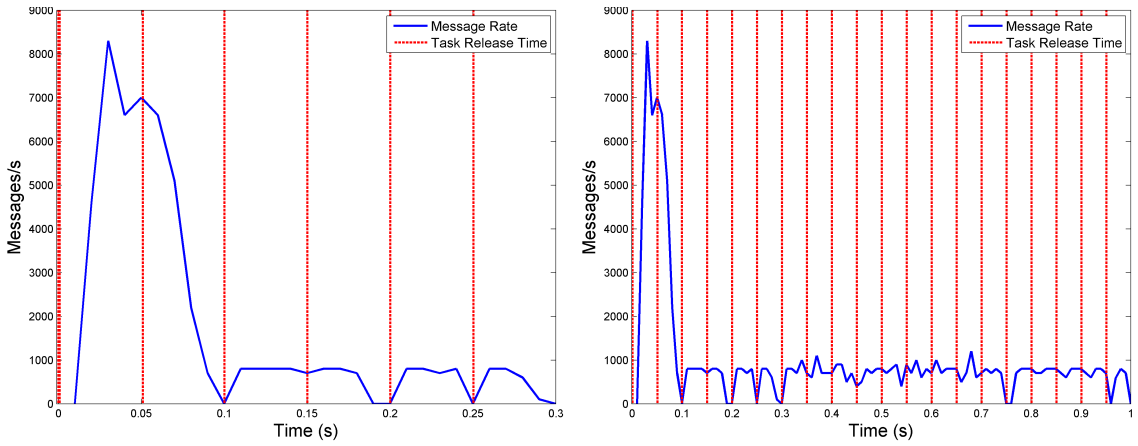
(b) Steady State Behavior

Figure 4-3: Message profile for a convergence sequence when 10 tasks seed the environment and pop-up tasks are released every 0.2s after

spike of messages (about 50) for a little less that 0.1 seconds. These few messages are used to assign the pop-up task to an agent quickly, then the global message state returns to idle. Part a) of Figure 4-3 shows a close up of the convergence for the first second of the demonstration, while part b) shows the general trend of the steady state behavior. For reference, the number of messages sent in the timespan shown in a) of Figure 4-3 was 620.

Given the results in Figure 4-3, we recognize that there may be a problem with defining agent convergence in terms of the global convergence if tasks are arriving at a rate faster than the algorithm requires to converge to a 0-message state. This case is shown in Figure 4-4, and is identical to the one outlined in Figure 4-3, except that tasks come every 50ms instead of 0.2s. This means that on the global scale, the network may never converge. This is an unavoidable side effect of allowing the network to handle pop-up tasks. As can be seen in part a) of Figure 4-4, only briefly does the global network ever converge. This can be seen even more compellingly in part b) of this figure: there are never large gaps, where the network has become completely quiet.

This shows that in general, it does not make sense to define the idea of global convergence for ACBBA. Despite this we can say that given pop-up tasks, the network



(a) Early Convergence

(b) Steady State Behavior

Figure 4-4: Message profile for a convergence sequence when 10 tasks seed the environment and pop-up tasks are released every 0.05s after

does not diverge. We can see the 10 tasks that are released at the beginning add a large number of messages to the network, but this spike is quickly handled and the message load stabilizes to the load of handling 1 new task at a time.

The above example highlights that in certain dynamic environments ACBBA convergence can only be thought of in terms of local convergence. This requires that each agent decide when their own bundles are conflict-free and ready for execution. Given the decentralized nature of ACBBA, the agents are free to make this decision independently of one another. In the example in Figure 4-4, the messages in the network during the steady-state phase are only due to a couple of agents deciding who wins the pop-up task. In general, 9 of the other 10 agents will not change their bundles based on the presence of this task, so when it is clear that they will not win it, they can decide that they have reached self convergence, even though there is no global convergence. In this way, certain agents can be in a converged state even though others are still actively engaging in a consensus phase.

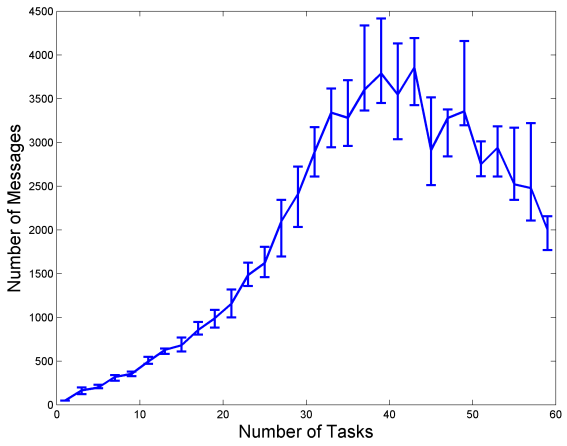
The parameter that ACBBA uses to define the notion of local convergence is called *separation time.* This number refers to the maximum amount of time in between the arrival of any two instances of relevant information. By relevant information, we mean any information that affects the local agents' ACBBA state. This means that any time

gap longer than a predefined upper bound that an agent hasn't received any relevant information, the agent can assume with high probability that it has reached local consensus. This separation time is computed in CBBA every time it checks to see if new information is available to update its bundle. A property of this parameter that can be seen in Figure 4-5 is that it is nearly independent of other relevant parameters. To explore this further we see a plot in part a) of Figure 4-5 of the number of messages versus number of tasks for a 10 agent simulation. Comparing part a) with part b), we can see that there is almost a direct mapping between convergence time and number of messages. However, for our convergence problem, the value for separation time is virtually independent to changes in number of messages, convergence time, or number of tasks in the network. In part c) of this figure we can see that it is also independent of the number of agents in the network. This value may not always be the same value given different communication constraints. However, since we observe it to be relatively independent of other parameters during CBBA operation, we can learn the value of this parameter and use it as a reliable measure of when each individual agent has converged.

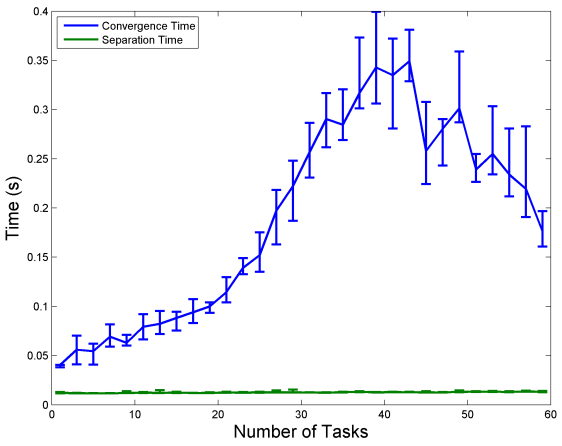
4.2 Asynchronous Replan

Another key feature that is enabled through ACCBA is the ability to execute what we call an asynchronous replan. Due to the nature of ACBBA's bundle building process, tasks in an agents' bundle can only be dropped if another agent outbids him for a task, or if a replan is called. A desired feature for a decentralized algorithm is for an agent to be able to react quickly to large changes in personal situational awareness. With the introduction of an asynchronous replan, an individual agent can decide to drop its entire bundle and rebuild it from scratch, so that it can re-optimize for these new local changes in situational awareness.

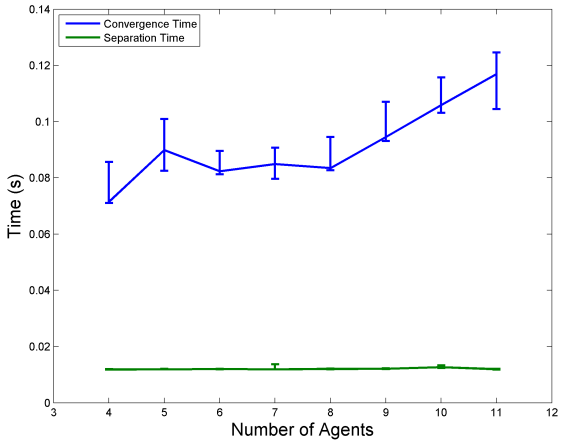
The notion of an asynchronous replan is especially attractive when the change to a decentralized agent's situational awareness will change that agent's bundle severely, but keep all the other agents' bundles unchanged. This is a typical scenario that might



(a) Number of messages



(b) Convergence time varying number of tasks



(c) Convergence time varying number of agents

Figure 4-5: Monte Carlo simulation results showing comparisons between number of messages, and convergence times

happen if a pop-up task is discovered near a remote agent. In some circumstances that agent will want to drop everything it is doing to service this task, but within the old CBBA framework, this would require a full global replan. In this situation, as well as many other possible situations in decentralized operations, events happen locally and only a few agents in a potentially large fleet care about. This asynchronous re-planning infrastructure gives these agents the flexibility to react dynamically, without tying the entire network down in a costly computational replan. These asynchronous replans utilize the same type of properties of ACBBA's convergence that normal convergence utilizes, such as the independence of separation time and the stability of

the global network to consensus disturbances.

4.3 The future of ACBBA

What ACBBA is able to effectively do is depart from the implicit global notions that the previous algorithmic operations suffered from. We are able to produce conflict free plans in very decentralized environments and the agents are able to execute missions with a real time executable task allocation algorithm. The algorithm looks closely at the information that is propagating through the network and is able to separate out the relevant information so only needed information is propagated through the network.

There are a few limitations to ACBBA that are being explored in current research. The first of these is the constraint on cost functions known as diminishing marginal gains. The set of cost functions that obey this property is unfortunately limiting and many approximations are needed to be made to create meaningful cost functions within this environment. Recent work that will be soon published has shown that cost functions can be used that break this diminishing marginal gains property as long as external bids follow DMG.

The second concern was introduced in chapter 3, and is the observation that it might make sense to combine implicit coordination with task consensus tools to come up with faster task allocations in highly constrained environments. Appendix A addresses this problem. The main idea is to utilize all available information to predict what other agents might want to do. This allows agents to predict task conflicts ahead of time and it increases the convergence rates of the CBBA and ACBBA algorithms.

Taking this idea to the extreme is what the authors have started calling Hybrid Information and Planning Consensus (HIPC). The purpose of this work is to capture all available information in order to come to consensus much faster even in environments with many inter-task constraints and inter-agent constraints. The three main focuses of this effort are :

1. **Agent Connectivity:** Communication environments can be sparse and unreli-

able. Planners that operate in these environments need to be utilizing all of the information that is available to create good cooperative plans under potentially poor communication. This includes creating maps of the likely local network conditions so that more complex consensus tools can be used.

2. **Decentralized Operations:** Truly decentralized operations rely on mostly local decision making. This usually involves dropping synchronization tools as well as giving up on much of the global behavior guarantees. The main push is to be able to internalize most of the decisions that each agent makes, while allowing for some localized cooperation through messaging. ACBBA has been a great start at this overall goal. Utilizing some of the advantages from focuses 1 and 3 can help augment ACBBA to make even more powerful decisions.

3. **Group Planning:** In a fleet of networked agents, if there is communication between agents, there is the possibility for more than just the task consensus information to be propagated. The past behaviors and decisions of other agents give clues to its location and priorities. With this extra information (along with other possible more direct information), agents can predict other agents behaviors in order to create overall more efficient algorithms.

Appendix A

Improving the convergence speed of the CBBA algorithm.

Through tests with a synchronized implementation of CBBA we have seen that there are significant communication delays when the algorithm is run in the field. This is due to many things (including but not limited to: poor network connections, inconsistent thread scheduling, heterogeneous computation platforms, etc.) but the main result is that each algorithmic iteration ends up taking a significant amount of time to synchronize. This "synchronization penalty" as it is often called in the literature is then multiplied by the number of iterations the algorithm takes to converge and becomes quite large even for medium sized problems. A potential solution was proposed that we call bootstrapping. The solution involves utilizing the fleet wide situational awareness already present on each of the agents to initialize the CBBA algorithm to a much better position than could typically be obtained with out this extra information. The goal is to add extra computation to the initial bids so that we can reduce the number of iterations throughout the rest of the algorithm and thus reduce the convergence time. This idea can be applied identically to ACBBA and it should be able to reduce the number of asynchronous iterations.

A.1 Implementation Details

CBBA with bootstrapping requires additional information called "situational awareness" in order to produce the convergence speed benefits. In many cases this additional situational awareness may only pertain to a subset of agents that are in local communication with each other. This situational awareness takes the form of positional estimates, health updates, capability constraints and other relevant state information about as many other agents as possible. At the start of a replan iteration, all of this information is acquired, and each agent then runs a centralized version on CBBA on board themselves. This centralized CBBA incorporates all of the information that it knows about other agents and creates predicted bids for all of the other agents that it has information about. While making these predicted bids for all of the other known agents, the centralized CBBA is able to anticipate conflicts and resolve them, without have to waste communication resources and introduce extra iterations. This is especially powerful when solving for coupled-constraints because it its possible to predict partners in cooperative endeavours, or predict if there will likely be no possible partnership. After complete bundles are produced, the agents then broadcast only the tasks that they have assigned to themselves. From this point onward, the algorithm proceeds exactly as nominal CBBA (or ACBBA), until a final assignment is reached. One of the primary reasons this approach was introduced was because it has relatively little effect on the rest of the machinery of CBBA but is able to make a noticeable improvement in the number of algorithmic iterations. Since this initialization is wrapped around the nominal CBBA, the algorithm retains all of its previous guaranteed convergence properties and performance compared to optimality.

A.2 Results and what they mean for performance

Tests were conducted with the bootstrapping algorithm to assess its ability to reduce the number of conflicts during the nominal algorithmic operation. Each of the plots shown below were conducted on 100 Monte Carlo runs. The blue lines above and

below the black ones are maximum and minimum bounds for the 100 test cases and the black line is the mean of all runs. For these results we choose a fairly dense environment of 100 tasks and 30 agents. The fuel cost and mission scores were such that every agent in the fleet had a non-zero incentive to service every task in the environment. This meant that there were a large number of conflicts and the planning problem was actually pretty hard.

Figure A-1 shows a plot of the number of total conflicts as a function of the fleet planned for. The vertical axis is scaled to 1 in an attempt to non-dimensionalize the test case. This plot will be broken up into its components below but one of the main intuitive results that it shows is roughly monotonic behavior when a higher percentage of the fleet is planned for. As you can see the environment was set up that on average, you must plan for every single agent in the fleet to guarantee a conflict free assignment. This is a very difficult assignment environment and is likely much more pessimistic environment than typical task allocation scenarios.

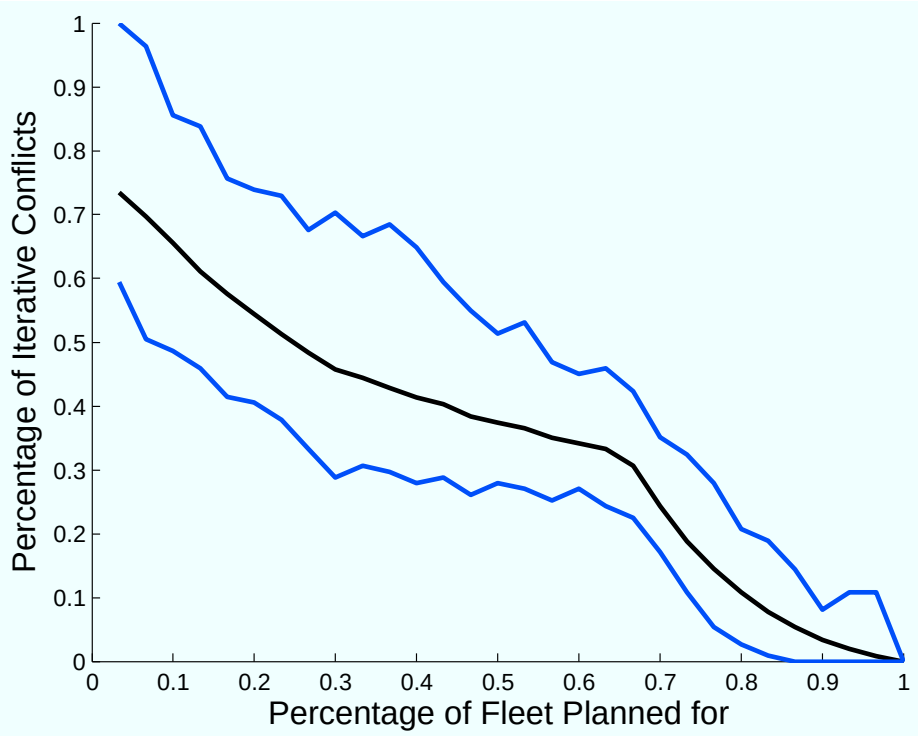


Figure A-1: This Plot shows the number of total conflicts as a function of the percentage of the fleet planned for.

The second figure (Figure A-2) shows a plot of the normalized number of initial conflicts as a function of the percentage of the fleet planned for. This translates to: how many conflicts do we see after the first iteration (or the bootstrapping consensus phase.) The furthest left data point on the plot corresponds to nominal CBBA and all other points moving the right are an increasing number of other agents planned for in the bootstrapping phase. This plot explicitly shows how difficult the fleet is to plan for because we don't see a large decay in the number of conflicts until we are planning for roughly 70% of the fleet. From here the plot decays quickly to 0. Figure A-3 shows

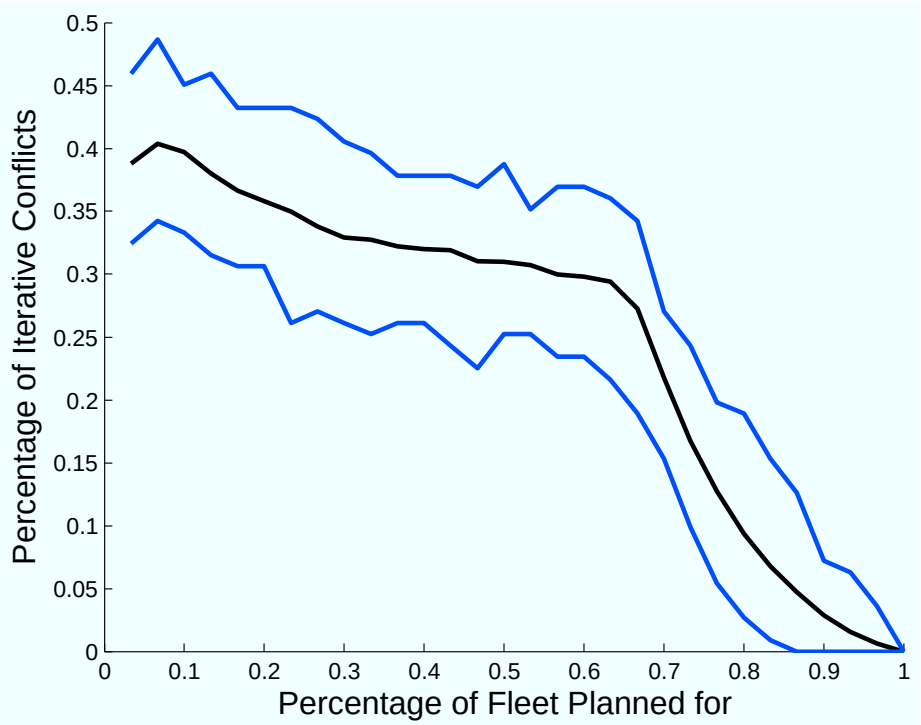


Figure A-2: This Plot shows the number of total conflicts as a function of the percentage of the fleet planned for.

the total number of conflicts that seen throughout the rest of the CBBA convergence process. This is the important figure because it is the graph that illustrates the reduction in the number of iterations because of the use of bootstrapping. (We don't explicitly plot iterations because this is a function of the network structure, be in fully connected networks the number of conflicts is roughly proportional to the number of iterations it takes for the algorithm to converge.) Again in the plot, the furthest left data point is the nominal CBBA solution and every point to the right is increasing the

number of other agents planned for. In the environment we worked in, after planning for 20% of the fleet initially, we see 50% less conflicts over the rest of the algorithm. And again, this environment is a very difficult planning environment, because of this a typical planning environment will likely be closer to the lower blue line where after 30% of the fleet is planned for, the number of conflicts is reduced by 90%. The main takeaway from this plot is that during the bootstrapping step, even though there may be quite a few conflicts in the first iteration, the algorithm is "learning" a lot about what tasks should be bid on. And after using this bootstrapping information, the rest of the convergence process is much faster. Bootstrapping is a quick and easy

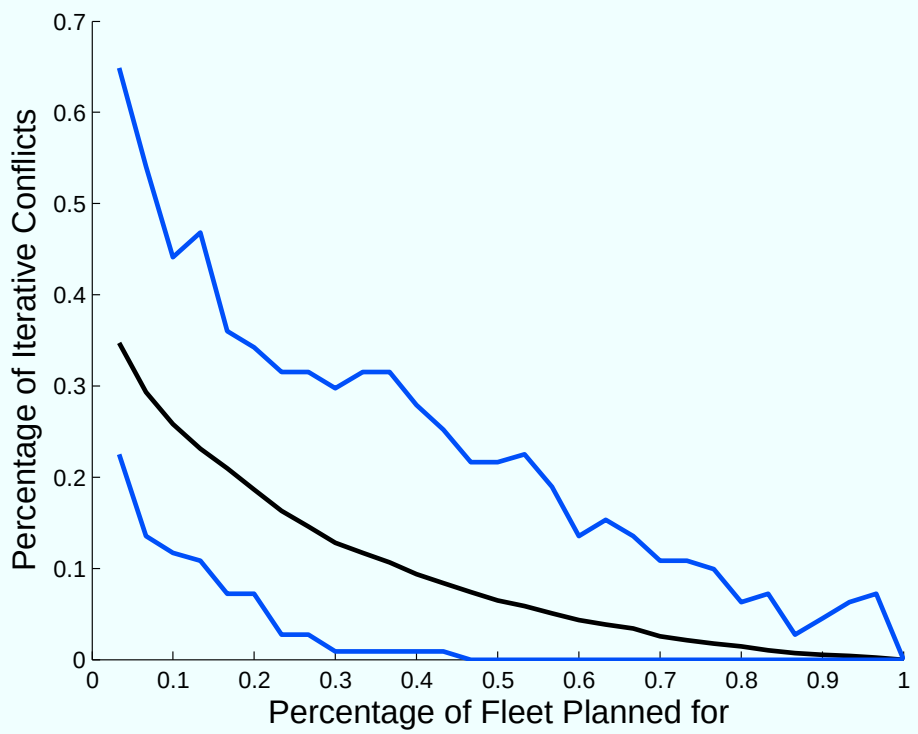


Figure A-3: This Plot shows the number of final conflicts as a function of the percentage of the fleet planned for.

incremental improvement that will decrease the convergence time for little extra work on the part of the planner.

Bibliography

[1] "Unmanned aircraft systems roadmap: 2007–2032," tech. rep., Office of the Secretary of Defense, 2007.

[2] United States Air Force Scientific Advisory Board, "Air force operations in urban environments – volume 1: Executive summary and annotated brief," Tech. Rep. SAB-TR-05-01, United States Air Force Scientific Advisory Board, `http://www.au.af.mil/au/awc/awcgate/sab/af_urban_ops_2005.pdf`, August 2005.

[3] U. S. A. F. Headquarters, "United States Air Force Unmanned Aircraft Systems Flight Plan 2009-2047," tech. rep., USAF, Washington DC, `http://www.govexec.com/pdfs/072309kp1.pdf`, 2009.

[4] M. L. Cummings, S. Bruni, S. Mercier, and P. J. Mitchell, "Automation architecture for single operator-multiple UAV command and control," *The International Command and Control Journal*, vol. 1, pp. 1–24, 2007.

[5] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, 1989.

[6] J. Bellingham, M. Tillerson, A. Richards, and J. P. How, "Multi-task allocation and path planning for cooperating UAVs," in *Cooperative Control: Models, Applications and Algorithms at the Conference on Coordination, Control and Optimization*, pp. 1–19, November 2001.

[7] C. Schumacher, P. Chandler, and S. Rasmussen, "Task allocation for wide area search munitions," in *American Control Conference (ACC)*, vol. 3, pp. 1917–1922, 2002.

[8] A. Casal, *Reconfiguration Planning for Modular Self-Reconfigurable Robots*. PhD thesis, Stanford University, Stanford, CA, 2002.

[9] Y. Jin, A. Minai, and M. Polycarpou, "Cooperative Real-Time Search and Task Allocation in UAV Teams," in *IEEE Conference on Decision and Control (CDC)*, vol. 1, pp. 7–12, 2003.

[10] L. Xu and U. Ozguner, "Battle management for unmanned aerial vehicles," in *IEEE Conference on Decision and Control (CDC)*, vol. 4, pp. 3585–3590, 9-12 Dec. 2003.

[11] D. Turra, L. Pollini, and M. Innocenti, "Fast unmanned vehicles task allocation with moving targets," in *IEEE Conference on Decision and Control (CDC)*, vol. 4, pp. 4280–4285, Dec 2004.

[12] M. Alighanbari, "Task assignment algorithms for teams of UAVs in dynamic environments," Master's thesis, Department of Aeronautics and Astronautics, Massachusetts Institute of Technology, 2004.

[13] T. M. McLain and R. W. Beard, "Coordination variables, coordination functions, and cooperative timing missions," *AIAA Journal on Guidance, Control, and Dynamics*, vol. 28, no. 1, 2005.

[14] D. A. Castanon and C. Wu, "Distributed algorithms for dynamic reassignment," in *IEEE Conference on Decision and Control (CDC)*, vol. 1, pp. 13–18, 9-12 Dec. 2003.

[15] J. Curtis and R. Murphey, "Simultaneous area search and task assignment for a team of cooperative agents," in *AIAA Guidance, Navigation, and Control Conference (GNC)*, 2003 (AIAA-2003-5584).

[16] T. Shima, S. J. Rasmussen, and P. Chandler, "UAV team decision and control using efficient collaborative estimation," in *American Control Conference (ACC)*, vol. 6, pp. 4107–4112, 8-10 June 2005.

[17] W. Ren, R. W. Beard, and D. B. Kingston, "Multi-agent Kalman consensus with relative uncertainty," in *American Control Conference (ACC)*, vol. 3, pp. 1865–1870, 8-10 June 2005.

[18] W. Ren and R. Beard, "Consensus seeking in multiagent systems under dynamically changing interaction topologies," *IEEE Transactions on Automatic Control*, vol. 50, pp. 655–661, May 2005.

[19] R. Olfati-Saber and R. M. Murray, "Consensus problems in networks of agents with switching topology and time-delays," *IEEE Transactions on Automatic Control*, vol. 49(9), pp. 1520–1533, 2004.

[20] M. Alighanbari, L. Bertuccelli, and J. How, "A Robust Approach to the UAV Task Assignment Problem," in *IEEE Conference on Decision and Control (CDC)*, pp. 5935–5940, 13–15 Dec. 2006.

[21] C. C. Moallemi and B. V. Roy, "Consensus propagation," *IEEE Transactions on Information Theory*, vol. 52(11), pp. 4753–4766, 2006.

[22] A. Olshevsky and J. N. Tsitsiklis, "Convergence speed in distributed consensus and averaging," in *IEEE Conference on Decision and Control (CDC)*, pp. 3387–3392, 2006.

[23] W. Ren, R. W. Beard, and E. M. Atkins, "Information consensus in multivehicle control," *IEEE Control Systems Magazine*, vol. 27(2), pp. 71–82, 2007.

[24] Y. Hatano and M. Mesbahi, "Agreement over random networks," *IEEE Transactions on Automatic Control*, vol. 50, pp. 1867–1872, Nov 2005.

[25] C. W. Wu, "Synchronization and convergence of linear dynamics in random directed networks," *IEEE Transactions on Automatic Control*, vol. 51, no. 7, pp. 1207–1210, 2006.

[26] A. Tahbaz-Salehi and A. Jadbabaie, "On consensus over random networks," in *44th Annual Allerton Conference*, 2006.

[27] M. Alighanbari and J. How, "Decentralized task assignment for unmanned aerial vehicles," in *IEEE Conference on Decision and Control (CDC)*, pp. 5668–5673, 12–15 Dec. 2005.

[28] S. Sariel and T. Balch, "Real time auction based allocation of tasks for multi-robot exploration problem in dynamic environments," in *Proceedings of the AIAA Workshop on Integrating Planning Into Scheduling*, 2005.

[29] A. Ahmed, A. Patel, T. Brown, M. Ham, M. Jang, and G. Agha, "Task assignment for a physical agent team via a dynamic forward/reverse auction mechanism," in *International Conference on Integration of Knowledge Intensive Multi-Agent Systems*, 2005.

[30] M. L. Atkinson, "Results analysis of using free market auctions to distribute control of UAVs," in *AIAA 3rd Unmanned Unlimited Technical Conference, Workshop and Exhibit*, 2004.

[31] T. Lemaire, R. Alami, and S. Lacroix, "A Distributed Task Allocation Scheme in Multi-UAV Context," in *IEEE International Conference on Robotics and Automation (ICRA)*, vol. 4, pp. 3622–3627, 2004.

[32] H.-L. Choi, L. Brunet, and J. P. How, "Consensus-based decentralized auctions for robust task allocation," *IEEE Transactions on Robotics*, vol. 25, pp. 912–926, August 2009.

[33] L. Brunet, "Consensus-based auctions for decentralized task assignments," Master's thesis, Department of Aeronautics and Astronautics, Massachusetts Institute of Technology, 2008.

[34] L. Brunet, H.-L. Choi, and J. P. How, "Consensus-based auction approaches for decentralized task assignment," in *AIAA Guidance, Navigation, and Control Conference (GNC)*, (Honolulu, HI), August 2008 (AIAA-2008-6839).

[35] L. Bertuccelli, H. Choi, P. Cho, and J. How, "Real-time Multi-UAV Task Assignment in Dynamic and Uncertain Environments," in *AIAA Guidance, Navigation, and Control Conference*, (AIAA 2009-5776) 2009.

[36] L. Fang, P. J. Antsaklis, and A. Tzimas, "Asynchronous consensus protocols: Preliminary results, simulations and open questions," in *Proceedings of the IEEE Conference on Decision and Control*, 2005.

[37] R. Olfati-Saber, A. Fax, and R. M. Murray, "Consensus and cooperation in networked multi-agent systems," *IEEE Transactions on Automatic Control*, vol. 95, pp. 215–233, January 2007.

[38] V. D. Blondel, J. M. Hendrickx, A. Olshevsky, and J. N. Tsitsiklis, "Convergence in multiagent coordination, consensus, and flocking," in *Proceedings of the IEEE Conference on Decision and Control*, 2005.

[39] A. Bemporad, M. Heemels, and M. Johansson, *Networked Control Systems*. Springer-Verlag Berlin Heidelberg, 2010.

[40] G. Mathews, H. Durrant-Whyte, and M. Prokopenko, "Asynchronous gradient-based optimisation for team decision making," in *Proceedings of the IEEE Conference on Decision and Control*, 2007.

[41] M. Cao, A. S. Morse, and B. D., "Agreeing asynchronously," *IEEE Transactions on Automatic Control*, vol. AC-53, pp. 1826–1838, September 2008.

[42] M. Mehyar, D. Spanos, J. Pongsajapan, S. H. Low, and R. M. Murray, "Distributed averaging on asynchronous communication networks," in *Proceedings of the IEEE Conference on Decision and Control*, 2005.

[43] M. M. Zavlanos and G. L. Pappas, "Dynamic assignment in distributed motion planning with local coordination," *IEEE Transactions on Robotics*, vol. 24, pp. 232–242, February 2008.

[44] M. M. Zavlanos, L. Spesivtsev, and G. J. Pappas, "A distributed auction algorithm for the assignment problem," in *Proceedings of the IEEE Conference on Decision and Control*, 2008.

[45] L. B. Johnson, S. Ponda, H.-L. Choi, and J. P. How, "Improving the efficiency of a decentralized tasking algorithm for UAV teams with asynchronous communications," in *AIAA Guidance, Navigation, and Control Conference (GNC)*, August 2010 (AIAA-2010-8421).

[46] A. K. Whitten, "Decentralized planning for autonomous agents cooperating in complex missions," Master's thesis, Massachusetts Institute of Technology, Department of Aeronautics and Astronautics, Cambridge MA, September 2010.

[47] D. C. Parkes and L. H. Ungar, "Iterative combinatorial auctions:theory and practice," in *Proceedings of the 17th National Conference on Artificial Intelligence*, 2000.

[48] A. Andersson, M. Tenhunen, and F. Ygge, "Integer programming for combinatorial auction winner determination," in *Proceedings of the Fourth International Conference on MultiAgent Systems*, 2000.

[49] S. de Vries and R. Vohra, "Combinatorial auctions: A survey," *INFORMS Journal of Computing*, vol. 15(3), pp. 284–309, 2003.

[50] S. Ponda, J. Redding, H.-L. Choi, J. P. How, M. A. Vavrina, and J. Vian, "Distributed change-constrained task allocation," in *American Control Conference (ACC)*, June 2012.

[51] S. Ponda, J. Redding, H.-L. Choi, J. P. How, M. A. Vavrina, and J. Vian, "Decentralized planning for complex missions with dynamic communication constraints," in *American Control Conference (ACC)*, (Baltimore, MD), July 2010.

[52] J. Bellingham, M. Tillerson, A. Richards, and J. How, "Multi-Task Allocation and Path Planning for Cooperating UAVs," in *Proceedings of Conference of Cooperative Control and Optimization*, Nov. 2001.

[53] M. Alighanbari and J. P. How, "Decentralized task assignment for unmanned aerial vehicles," in *IEEE Conference on Decision and Control and European Control Conference (CDC-ECC '05)*, 2005.