Automated Mission Parallelization for Unmanned Underwater Vehicles

Gary Giger, Mahmut Kandemir
CSE Department
The Pennsylvania State University
University Park, PA 16802, USA
{giger, kandemir}@cse.psu.edu

S. Dan Lovell, John Dzielski
Unmanned Vehicle Systems Department
Applied Research Laboratory/PSU
University Park, PA 16802
{sdl12, jed}@enterprise.arl.psu.edu

Sekhar Tangirala
Lockheed Martin Company
Maritime Systems and Sensors
Riviera Beach, FL 33404
sekhar.tangirala@lmco.com

Abstract

In the past Unmanned Underwater Vehicles (UUVs) have been used to perform different tasks such as mapping the ocean floor and on-site exploration. Current research reveals even more applications for UUVs in many different areas including the military, scientific research, commercial uses, and even certain aspects of law enforcement. Unless missions for these applications can be written using a high-level programming language, many of these missions will be error prone and difficult to maintain, especially if missions are created for a group of cooperating UUVs to achieve a common goal. If the group contains UUVs that have different capabilities all together, writing and maintaining multiple missions for this heterogeneous group will be even more difficult. The goal in this paper is to propose methods that utilize an existing high-level Mission Programming Language (MPL) as a framework to allow a user to create a set of parallel sub-missions for a group of UUVs from a user specified mission. We discuss various methods of generating automatically a set of parallel sub-missions from this user specified mission and provide some preliminary results from our simulations that implement the proposed methods.

1. Introduction

In the past Unmanned Underwater Vehicles (UUVs) have been used to perform tasks such as mapping the ocean floor and surveying wreckage such as sunken ships. As UUV research continues, more and more applications for UUVs are being proposed in many different areas including mine detection and identification (Freitag et al 2005), environmental monitoring of pollutant levels found in water as well as observing and collecting data on aquatic life (Tripp 2006). Even certain aspects of law enforcement are considering using UUVs to patrol commercial fishing areas to enforce established fishing regulations (Tripp 2006). Some of these applications even include using multiple cooperating UUVs.

Since many UUV missions are often critical, successfully programming missions for these vehicles still remains a key factor in a particular mission's success. As with writing any type of program in a specific language, errors may be introduced due to typos, incorrect calculations, or simply the user's lack of knowledge of the particular language. Furthermore, with research moving towards using groups of UUVs to accomplish a common goal (with larger and more complex mission files), manually programming missions for multiple vehicles could introduce even more errors, especially if the vehicles used in the mission are heterogeneous, that is, they each use a different language or possess different capabilities (e.g., having different types of sensors for collecting various types of data). This is why high-level Mission Programming Languages (MPLs) such as those mentioned in (Giger *et al* 2006), (Eberbach *et al* 2003), (Duarte *et al* 2005), (Davis 2005) have been developed to help reduce many of these errors. While high-level MPLs are very useful to program missions for a single UUV, they lack the capability to process a mission for a group of UUVs. In the current setting, the user must write a separate mission for each vehicle in the group. This can make the user's job even more burdensome, especially if the group of UUVs is heterogeneous.

There has not been much prior work in using a high-level MPL for UUVs to generate a set of parallel sub-missions for a group of vehicles. Related work has been done in (Eberbach *et al* 2003), (Duarte *et al* 2005) to allow a group of heterogeneous UUVs to communicate using a Common Control Language (CCL) that also includes a supporting compiler and interpreter, but a user must still create missions and take into account any parallel aspects that is to be executed by more than one vehicle. Our goal in this paper is to extend the previous work on our MPL and propose methods to enable such a language to process a user specified mission and generate automatically a corresponding set of parallel sub-missions for a particular group of UUVs.

The rest of this paper is organized as follows. Section 2 briefly introduces the idea of UUVs. Section 3 presents an overview of our high-level MPL, the mission controller that uses this language and the compiler for this MPL. Section 4 discusses the concept of a parallel mission and provides motivation for executing a mission in parallel using multiple UUVs. Section 5 illustrates our proposed algorithms for generating a set of parallel sub-missions for a group of UUVs followed by Section 6, which provides simulations results regarding these proposed techniques. Section 7 discusses our conclusions and future work.

2. Overview of Unmanned Underwater Vehicles

A UUV is an unmanned vehicle that is designed and built to carryout specific tasks, often referred to as missions, in an underwater environment. UUVs have computer hardware running controller software, called a Mission Controller (MC), which is installed to command the vehicle. This controller software executes *orders* (mission commands) contained within a mission in the same manner instructions of a computer program are executed. Orders are often executed sequentially, and sometimes

concurrently (based on how the Mission Controller is structured) so that the UUV can achieve the overall goal of the mission. Common orders for an UUV often include waypoint orders, orders for taking GPS readings, orders to operate any equipment or devices that the vehicle might carry, and survey orders for some of the more sophisticated UUVs. Missions are typically written in a language that was developed for the particular UUV to allow an operator to create these missions rather than programming using a low-level language (i.e. language similar to assembly). After a mission is written, it is loaded onto the UUV to be executed without any operator intervention once the UUV is deployed.

3. High-Level Mission Programming

3.1 Language

A UUV mission is typically constructed using a set of orders. UUV missions written by hand in a low-level language can sometimes be hard to understand. This can result in untidy code that could contain software bugs, which can make it very difficult to maintain these missions. Since many UUV missions are critical, high-level MPLs were developed in order to make it easier for the operator to create missions and reduce the number of programming errors. These languages also support high-level constructs that include looping and conditional statements, which allows these languages to handle complex and challenging missions. These constructs are what make a high-level MPL similar to general high-level languages such as C++ and Java. As with any programming language, high-level MPLs have a language specification that shows how to use the different orders. A sample of the MPLs specification for a survey order in our high-level MPL is shown in Figure 1. A survey order is characterized by listing a set of four points the makes up a survey region. In this specification in Figure 1, the elements marked as *Critical* are required elements. They need to be explicitly specified by the user. The elements marked as *Optional* are the elements that, if omitted, the compiler will substitute the default values (enclosed in parenthesis) for them. More details regarding our high-level MPL are discussed in Giger *et al* 2006.

3.2 Mission Controller and Compiler

The Mission Controller (MC) (Tangirala *et al* 2005) is the controller software that is installed on the UUVs computer hardware and it is responsible for executing the orders contained within the UUV mission. In our case, the MC is a three-tiered system of interacting hybrid modules, as depicted in Figure 2. Modules at any level may issue commands to other modules at the same level and to any level below. A module can send a response to any module in the current level and to any level above. All communications take place through synchronized events and shared data in a block that is accessible by all the modules in the MC. The highest level contains the Coordinators, which are responsible for scheduling mission orders contained within the user specified mission to achieve the overall goal of the mission. At the lowest level are the Behavioral Controllers (BCs) which, interface directly with the Vehicle Controllers (VCs) through the Mission Controller to Vehicle Controller (MC2VC) and the Vehicle Controller to Mission Controller (VC2MC) interfaces. The VCs are responsible for controlling specific hardware in the UUV itself. As the UUV completes specific orders, responses from these completed orders are sent by the UUV to the MC via the VC2MC interface up the chain of command to the coordinators. The coordinators then issue the next order and this process continues until the overall goal of the user specified mission is completed.

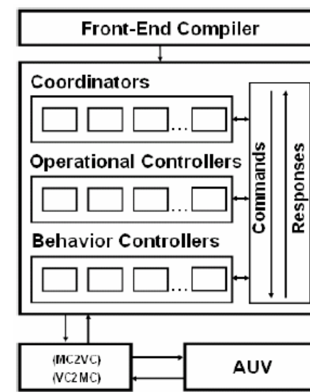

Figure 2 – Mission Controller Architecture

```
Start                 Survey_Order
TopLeft_Lat           Critical Float Deg / Rad
TopLeft_Long          Critical Float Deg / Rad
BottomRight_Lat       Critical Float Deg / Rad
BottomRight_Long      Critical Float Deg / Rad
Survey_Depth          Critical Float M / Ft
Survey_Speed          Critical Float Mps / Knots
SwathWidth            Optional Float (200.0) M / Ft (M)
```

Figure 1 - Sample of the language specification for a survey order in the MPL. Note that this specification has been simplified for the sake of clarity.

The MC also interacts with a compiler that is used to compile and verify the user specified mission so that it can be loaded onto the mission controller for execution. This compiler is located on top of the MC Figure 2. An example user specified mission and the use of this compiler and MC is discussed in detail in our technical report (Giger *et al* 2007).

4. Parallel Missions

4.1 Motivation

With current research moving towards using multiple UUVs to accomplish a common mission (Eberbach *et al* 2003), (Duarte *et al* 2005), (Davis 2005), the power and flexibility will be greater than ever before for many UUV applications. For instance, a single survey order could be divided across multiple vehicles so that the entire area is surveyed in parallel, thus, taking less time when compared to the case where only one vehicle is used. This simple example demonstrates the potential for parallel, coordinated operations in which each vehicle is assigned a small slice of the entire survey area. However, in order for a mission involving multiple UUVs to be effective, these missions will need to be divided among the multiple vehicles in such a way so that they are utilized to their fullest potential.

A high-level MPL is suitable to handle missions for a single UUV. But what if multiple UUVs were required to execute a specific mission as illustrated in this previous scenario? Without any automated parallelization help, when a group of UUVs were to execute a particular mission such as a hydrographic survey, the operator would need to determine the number of vehicles that will be available at the time when the mission will be executed, then manually divide the mission into the correct number of submissions. Manual calculations must be done to divide the original survey order into the correct number of sub-missions, which could easily introduce mathematical errors. A person who is a programmer by trade and who is familiar with concepts relating to parallel sub-missions could potentially create a set of parallel missions for this original survey order with a minimal amount of errors. Knowing the number of vehicles and their types beforehand, this person should be able to create a set of parallel sub-missions, which would involve dividing the original survey order into sub pieces and manually assigning them to a set of UUVs. However, the people who create missions for UUVs may not be particularly good at programming multiple UUVs. In fact, it might be very difficult for an average scientist to write parallel missions involving multiple, heterogeneous UUVs. The idea of parallel missions might not be understood conceptually by these people, which could result in a set of parallel sub-missions that contain errors. As a result, the capabilities of the group of UUVs may not fully be utilized and this could result in improperly collected data. This is why an automated method is needed for generating these parallel sub-missions. This will remove the responsibility of creating these parallel sub-missions out of the operator's hands and assign this job to the compiler, which will require additional functionality to handle the splitting of orders. We will now discuss modifications to our MPL in order to support this idea of a parallel mission.

4.2 Parallel Regions

In order to solve this problem of automatically generating a set of parallel sub-missions, we added a new construct known as a *Parallel Region* to our existing MPL. A user adds orders to this parallel region within the mission as shown in Figure 3. In this example, a typical survey order is contained within the parallel region denoted by the reserved words *Parallel_Region_Start* and *Parallel_Region_End*. These keywords act similar to the *pragma* keyword (Barclay 1990) in C. Any orders contained within this parallel region will be understood by the compiler to generate a set of parallel sub-missions based on the original orders. Even though the responsibility is placed on the user to indicate which orders need to be executed in parallel, this is the only responsibility the user has during the process of generating a set of parallel sub-missions, which leaves the rest of the work to be done by the compiler. This allows a user to write a mission using the high-level MPL without having to rely on any vehicle specific data before hand such as the specific number of available vehicles and their types.

```
Parallel_Region_Start
{
    ...

    Start_Order   : Survey Order
    TopLeft_Lat   : +23.000 Deg
    TopLeft_Long  : -70.000 Deg
    BotRight_Lat  : +23.500 Deg
    BotRight_Long : -70.500 Deg

    ...
}
Parallel_Region_End
```

Figure 3 - Sample code illustrating the idea of a parallel region. Here, the parallel region contains a survey order based on the language specification from Figure 1**.**

Parallel regions are also transparent, that is, if the user were to remove the parallel region keywords, the compiler will simply treat all of the orders contained in the former parallel region as a set of sequential orders. In other words, if no parallel region is encountered by the compiler, it assumes that there is only one UUV available for the user specified mission. This feature makes it very easy to transform an existing mission into a parallel mission for multiple UUVs.

The most important point to emphasize is that the compiler, when given an objective, has the flexibility to decide how to generate a set of parallel sub-missions for a group of vehicles from a parallel region construct. (Note the vehicle information such as the different types of UUVs and the number of each type is read by the compiler from a database). For example, if time is a constraint, the compiler can choose to assign the appropriate vehicles to the survey order so this order can be executed in parallel within the specified time constraint, thus reducing the execution time. On the other hand if vehicle usage is a constraint, the compiler can select the minimal number of vehicles that still possess the required capabilities to satisfy the survey order. We now discuss our proposed methods that are used to automatically split a user specified mission (using the parallel region construct) into a set of parallel sub-missions.

5. Our Partitioning Algorithms

In this section we propose two algorithms that, when given a user specified mission, take into account the available number of vehicles, their types, and the set of requirements for the mission, and generate a set of sub-missions. We discuss these algorithms in detail, including how this data is used to split a user specified mission into the appropriate set of parallel sub-missions for a survey order. (Note that while a parallel region can contain any type of order, our current implementation considers only the survey orders for parallel execution across multiple UUVs). Section 5.1 discusses an algorithm for reducing the amount of time it takes to complete a mission. Section 5.2 presents an algorithm that reduces the number of vehicles used in a mission.

5.1 Reducing Execution Latency under Vehicle Bounds

Our first algorithm allocates available UUVs to the survey orders in a parallel region to minimize the overall mission execution latency. When the compiler must generate a set of parallel sub-missions, it needs to take into consideration the available vehicle data such as the different vehicles types and the number of each vehicle type. Let us now consider the example of a rec-

Table 1 - Vehicle data for the example in Figure 3

Vehicle Type	Sensors	Vehicle Count
Vehicle 1 (V_1)	{S_1, S_2}	4
Vehicle 2 (V_2)	{S_1}	2
Vehicle 3 (V_3)	{S_1, S_4}	4
Vehicle 4 (V_4)	{S_3}	2

tangular region that needs to be surveyed where the entire area requires both sensors S1 and S2. This survey area is assumed to be specified using a single survey order contained within a parallel region similar to the example shown in Figure 3. The available vehicles for this example are shown in Table 1. In order to reduce the amount of time to execute the mission, we will need to consider all of the vehicles that can be used to satisfy this survey area. Note that vehicle types V_1, V_2, and V_3 can be used to fulfill this survey area as these vehicles contain sensors S_1 and/or S_2. Thus, the total number of vehicles that can be allocated to this order is ten. Since vehicle type V_1 is the only type that contains sensor S_2, the four vehicles of this type can be used to satisfy the S_2 sensor requirement of the region. The remaining vehicles can then be used to satisfy sensor requirement S_1 for this region. Note that there is no sensor requirement for sensor S_3, thus the two vehicles with sensor S_3 remain idle. After taking this information into consideration, our algorithm generates the parallel sub-missions illustrated in Figure 4. Our algorithm for this approach is shown in Figure 5, which is oriented towards reducing the amount of time used to complete a Survey Order.

To briefly summarize this algorithm, the survey order (line 1) is retrieved from the parallel region of the user specified mission. The vehicle types and number of each type (line 2) are then retrieved from the available data source. The list of vehicle types that are capable of satisfying particular regions of the survey order are determined (line 3). The list of regions are then sorted (line 4) from largest to smallest based on the number of vehicles that were assigned to each particular region. The idea here is to survey the regions that require the most vehicles first so that more vehicles are available to handle later regions that require fewer vehicles. Lines 5 through 11 generate the set of parallel sub-missions based on the number of each vehicle type for each region. This algorithm uses a greedy approach (line 7) such that it chooses all of the vehicles capable of satisfying the current instance of a particular region. It may therefore produce

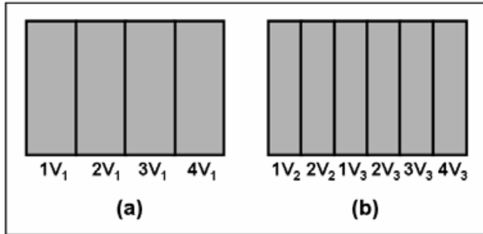

Figure 4 - Survey order split into four sub-missions and assigned to four vehicles of type V_1 to satisfy sensor requirement S2. (b) Same survey order split into six sub-missions and assigned to two vehicles of type V_2 and four vehicles of type V_3 to satisfy sensor requirement S1. All vehicles assigned to this survey order can execute each sub-mission in parallel.

```
REDUCE-SURVEY-TIME
1  SA ← get_survey_areas ()       // Provided from user mission
2  VT ← get_vehicle_types ()      // Retrieved from data source
3  Vehicle_List . Types = allocate_vehicle_types ( SA, VT )
4  sort_satisfy_list (Vehicle_List ) // Largest to smallest
5  for j = 1 to size_of ( SA )
6    for i = 1 to size_of ( Vehicle_List_j )
7      num_slices ← get_vehicle_count_for_subregion( Vehicle_List_j .Type_i )
8      slice_width ← calc_width_of_subregion ( SA_j )
9      create num_slices of parallel sub missions for SA_j using slice_width
10   next i
11 next j
```

Figure 5 - Algorithm for reducing the amount of time to execute a parallel region.(a)

a suboptimal solution. For more details regarding this example and an additional example showing a more complex set of regions, please refer to our technical report (Giger *et al* 2007).

An important point to note about overlapping regions captured by a given parallel region construct is that no work is duplicated if the overlapping portions of multiple regions require the same set of sensors. For example, if the portions of two regions overlap and both require sensor S1. The overlapping portions requiring sensor S1 will only be covered once. Thus, if the survey for region 2 is executed first, its overlapping portion with region 1 requiring sensor S1 is covered during the pass over region 2 and not a second time when the non-overlapping portions of region 1 are surveyed.

5.2 Reducing the Number of Vehicles Used

Reducing the number of vehicles used to complete a particular survey order rather than reducing the amount of time is another important problem to consider when dealing with missions for multiple UUVs. Consider the same example from Figure 3 where again the survey area requires sensors S1 and S2 and the available vehicles are shown in Table 1. Our objective is to select the minimum number of vehicles while at the same time satisfying all of the sensor requirements for each point in the specified area. To solve this problem, we adopted the set covering algorithm (Corman 2001), which is an optimization algorithm that is used to solve resource allocation problems. It attempts to find the minimal number of sub-sets that can cover all of the points in a given area. In the set covering problem, we are given a universal set U and the sets $s_1,..,s_m$. The objective is to cover all elements of U with as few s_i sets as possible. Given an arbitrary instance of set covering, we can define an equivalent instance of our problem. Our algorithm for the set covering problem is shown in Figure 6, which makes use of a greedy solution. In our context, U will be the set of points requiring specific sensors (line 1) and s_i will be the set of sensors

```
GREEDY-SET-COVER (X, S)
1 U ← X
2 C ← Ø
3 while U ≠ Ø do
4    select sj ∈ S that maximizes | s_j ∩ U |
5    C ← C ∪ s_j
6    U ← U ∩ s_j
7 end while
8 return C
```

Figure 6 - Greedy algorithm for the vehicle minimization problem.

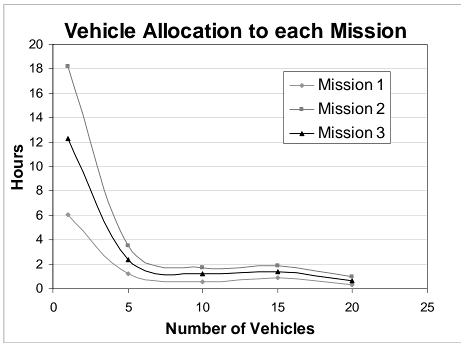

Figure 7 - Mission execution time versus the number of vehicles used for a particular mission.

equipped on vehicle v_i where $s_i \in S$ and $v_i \in V$. A single vehicle can visit all points where its sensors are required. For this reason, we cannot approximate our problem better than the set covering problem, and therefore, we must settle for $\Omega(\log n)$ approximation ratio (Feige 1998) where n is the number of points in the survey area or simply the number of points in U. That is, we model our problem as follows. U is the set of pairs (P_j, S_i) such that point P_j requires sensor S_i. If T is a set of points a vehicle V_i can visit on a single trip, we can define set $A = \{(P, S) \in U : P \in A \,\&\, S \in set(v)\}$.

6. Simulation Results

We obtained our results by adding the algorithms from section 5 to the existing compiler in the MC. We then created a set of missions and used this compiler to automatically generate the set of corresponding parallel sub-missions for each mission. Once the missions were generated, they were loaded onto the mission controller for execution. To simulate the MC, we use a tool known as Teja NP (http://www.teja.com), which provides the capability for modeling parallel tasks. We can run multiple instances of the MC using Teja NP where each instance simulates an instance of a UUV.

6.1 Execution Latency Simulation

We ran several test cases for our parallelization approach discussed in Section 5.1. We used three different missions for this simulation (Figure 8). For each mission we ran the algorithm from Section 5.1 using 1, 5, 10, 15, and 20 UUVs for. Note that

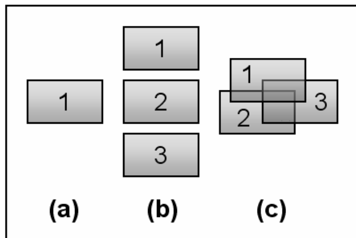

Figure 8 - (a) Single survey area (b) Three survey areas where each requires the same sensor type. (c) Same three survey areas, this time they overlap with one another.

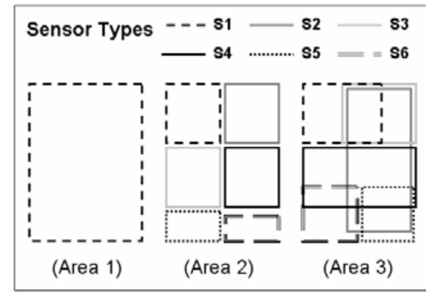

Figure 9 - (a) Survey Area 1, which only requires one sensor. (b) Survey Area 2 containing sub-regions each requiring a different sensor reading. (c) Survey Area 3 containing overlapping sub regions each requiring multiple sensor readings.

for this set of simulations, we used a set of vehicles all of the same type and survey areas that were equipped all with the same sensor types. The results of this simulation are shown in Figure 8, which shows the time it takes to execute the missions from Figure 8.

Mission 1 took the least amount of time in all cases since there were multiple vehicles all working on a single survey order. Mission 2 took the longest time since these areas were independent of one another. Mission 3, on the other hand, consisted of the same survey areas as in Mission 2, but these areas overlapped with one another. The time to complete this mission was less when compared to Mission 2. This is due to the lack of duplication of work when overlapping regions require the same sensor as mentioned before.

6.2 Minimal Number of Vehicles Simulation

Next we performed simulations with the algorithm proposed in Section 5.2 with the missions shown in Figure 9 using the vehicles listed in Table 2. Note that the different areas were surveyed using each of the vehicle groups from Table 2. Figure 10 provides the results of these simulations. For example, in using Group B for Survey Area 2, our set-covering based algorithm used one of each type of vehicle from Group B to survey each region that requires a different sensor, thus using only 6 vehicles and leaving the other 4 free. Notice the vehicles in Group A, which contains all sensors, uses one vehicle for each of the different survey areas. Group B only uses one vehicle for Area

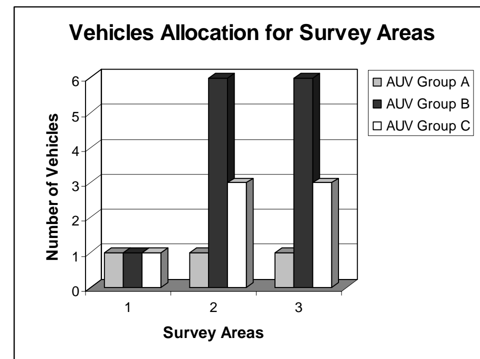

Figure 10 - Number of vehicles allocated for a survey area using our set covering algorithm from Figure 6.

Table 2 – Vehicles used in simulations in section 6.2

Group	Types	Sensors	Vehicle
A	V_1	S_1 S_2 S_3 S_4 S_5 S_6	10
B	V_2	S_1	2
	V_3	S_2	2
	V_4	S_3	2
	V_5	S_4	2
	V_6	S_5	1
	V_7	S_6	1
C	V_8	S_1 S_2	2
	V_9	S_1 S_2 S_3	2
	V_{10}	S_2 S_3 S_4	3
	V_{11}	S_5 S_5	3

1 since our algorithm picked the one vehicle that satisfied the sensor requirement for that particular area. On the other hand Group B, where each vehicle type only contains one type of sensor, used 6 vehicles for both survey areas 2 and 3. Group C has a mix of different sensors and used only 3 vehicles for areas 2 and 3. As mentioned in the previous section, this allocation method is by no means optimal, but it still does provide a reduced number of vehicles for each survey area. In fact, in this set of simulations our algorithm used a maximum of only 6 UUVs for any of the survey areas depicted in Figure 9.

7. Conclusion and Future Work

This paper has presented initial results from our efforts on building mission parallelization tools for a group of UUVs. We considered two algorithms in this work: one for reducing the mission execution latency and the other for reducing the number of vehicles used. The experimental results collected so far are encouraging. Our future work not only includes enhancing our proposed algorithms from this paper, we are also considering other approaches as well. We are exploring an integer linear programming solution from the domain of operations research (Winston 2004) that will include multiple constraints and will provide the user with additional mission objectives such as reducing power consumption for the group of vehicles. We are also looking at developing a graphical mission tool that will allow an operator to create missions in a point and click fashion and further reduce the amount of code an operator actually writes. Overall, we believe our compiler based approach for generating a set of sub-missions for a group of cooperating UUVs is a step toward actually realizing multi-UUV missions in both military and civilian domains.

References

Barclay, K. A. 1990. *ANSI C Problem Solving and Programming*. Prentice Hall International.

Corman, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. 2001. *Introduction to Algorithms Second Edition*. The MIT Press.

Davis, D. T. 2005. Automated Parsing and Conversion of Vehicle-Specific Data into Autonomous Vehicle Control Language (AVCL) using Context Free Grammars and XML Data Binding. In the 14th International Symposium on Unmanned Untethered Submersible Technology. Durham, NH.

Duarte, C. N., Buzzell, C., Martel, G. R., Crimmins, D., Komerska, R., Mupparapu, S., Chappell, S., Blidberg, D. R., and Nitzel, R. 2005. A Common Control Language to Support Multiple Cooperating AUVS. In the 14th International Symposium on Unmanned Untethered Submersible Technology. Durham, NH.

Eberbach, E., Duarte, C., Buzzell, C., and Martel, G. 2003. A Portable Language for Control of Multiple Autonomous Vehicles and Distributed Problem Solving. In Proceedings of the 2nd International Conference on Computational Intelligence, Robotics and Autonomous Systems. Singapore.

Feige, U. 1998. A threshold of ln n for approximating set cover. J. ACM 45, 4 (Jul. 1998), 634-652.

Freitag, L., Grund, M., von Alt, C., Stokey, R., and Austin, T. 2005. A Shallow Water Acoustic Network for Mine Countermeasures with Autonomous Underwater Vehicles. IEEE Oceans Conference. Washington DC.

Giger G., Kandemir, M., and Lovell, S. D. 2007. Automatic Generation of Parallel Missions for Autonomous Underwater Vehicles, Technical Report, CSE 07-006, Dept. of Computer Science and Engineering, The Pennsylvania State University.

Giger, G., Xue, L., Tangirala, S., and Kandemir, M. 2006. High Level Mission Programming Support for Autonomous Underwater Vehicles. AUVSI's Unmanned Systems North America. Orlando, FL.

Tangirala, S., Kumar, R., Bhattacharyya, S., O'Connor, M., and Holloway, L. E. 2005. Hybrid-Model based Hierarchical Mission Control Architecture for Autonomous Underwater Vehicles. In Proceedings of the 2005 American Control Conference. Portland, OR.

Tripp, S. T. 2006. Autonomous Underwater Vehicles (AUVs): A Look at Coast Guard Needs to Close Performance Gaps and Enhance Current Mission Performance, Technical Report, ADA450814, Coast Guard Research and Development Center, Groton, CT.

Winston, W. L. 2004. *Operations Research Applications and Algorithms*. Brooks/Cole.