Procedia Computer Science

Volume 29, 2014, Pages 1277–1288

ICCS 2014. 14th International Conference on Computational Science

Optimizing Dynamic Resource Allocation

Lucas W. Krakow, Louis Rabiet, Yun Zou, Guillaume Iooss,
Edwin K. P. Chong, and Sanjay Rajopadhye

Colorado State University, Fort Collins, Colorado, U.S.A
Lucas.Krakow@colostate.edu, Louis.Rabiet@colostate.edu, YunZou.colostate@gmail.com,
Guillaume.Iooss@gmail.com, Edwin.Chong@colostate.edu, Sanjay.Rajopadhye@colostate.edu

Abstract

We present a formulation, solution method, and program acceleration techniques for two dynamic control scenarios, both with the common goal of optimizing resource allocations. These approaches allocate resources in a non-myopic way, accounting for long-term impacts of current control decisions via nominal belief-state optimization (NBO). In both scenarios, the solution techniques are parallelized for reduced execution time. A novel aspect is included in the second scenario: dynamically allocating the computational resources in an online fashion which is made possible through constant aspect ratio tiling (CART).

Keywords: automatic parallelization, constant aspect ratio tiling (CART), high performance computing, nominal belief state optimization (NBO), optimal Stochastic Control, partially observable Markov decision process (POMDP), polyhedral model, stochastic dynamic programming, target tracking, unmanned aerial vehicles (UAVs)

1 Introduction

With the continued progress of technology making smaller and more economical sensing platforms, and the accompanying increase in computational capabilities, there is a drive to selectively acquire and process the mass amounts of data collected in sensor networks. Target tracking sensor networks are a prominent setting for these types of considerations. The overlying term for the control and application of the system components is *resource management*. In particular, *sensor* resource management has experienced a pointed increase in interest corresponding to smaller less expensive sensors, often focusing on increased network lifespan, lowering failure rates, and energy consumption. The control of such sensors is an inherently dynamic data-driven task, where the control algorithms need to manage dynamic changes to the targets, the sensors and environmental conditions. It therefore benefits from the DDDAS paradigm.

Mounting the sensors themselves on articulating or fully mobile platforms allows the dynamic positioning of the sensors or their field of view (FoV). This was the subject of our initial study,

This work was supported in part by AFOSR under contract FA9550-13-1-0064.

doi:10.1016/j.procs.2014.05.115

where we examine sensors mounted on unmanned aerial vehicles (UAVs). By controlling the acceleration of the UAVs, we determine the sensor placements and thus the subject and quality of their measurements. The ability to issue these commands to the UAVs forms closed-loop feedback control, where control decisions are based on the minimization of the tracking error, estimated indirectly as a function of the observations.

We formulate a partially observable Markov decision process representing the UAV control scenario and apply a novel solution approach, namely nominal belief-state optimization (NBO) [9]. Although NBO provides a large computational reduction over other non-myopic dynamic control algorithms, as the complexity of the scenarios grow, the computational load still becomes burdensome when considering real-time control. For this reason, our initial study chose to port portions of the control algorithm to parallelized code for Graphics Processing Unit (GPU) acceleration. The translation of this code was performed by manual code optimization, guided by the principles of program analysis and the polyhedral model [13]. While the results summarized in Section 2 are satisfactory for computational speed-up, the prototype code was only viable for a very specific high performance computing (HPC) platform, an NVIDIA GTX 480 class of GPU. Given this, the concept of producing code for the progressing multi-core platforms would be prohibitively labor intensive.

Motivated by the need to develop a portable methodology in the context of rapidly evolving architectures, and applications, we desire to systematically, preferably automatically, map our algorithms to additional platforms in an optimal manner. In this context, the parallelization portability is critical, and manual code development is not a viable option. This can be accomplished via automatic parallelization in the polyhedral model, a mathematical formalism for parallelization of a restricted but widely applicable class of programs [13, 5, 6]. Numerous research tools are now available for automatic parallelization of imperative loop and/or functional and equational programs [15, 2]. Our work is based on the `AlphaZ` tool using the equational language `Alpha`. Current extensions including subsystems and adaptations to irregular amorphous computations provide an avenue for its use in our scenarios.

This choice now allows a single code base to generate executable code for a multitude of computing platforms and configurations. With these additional possibilities it becomes apparent, our "system" is expanding, providing more points of control. In certain situations it is evident that managing the resources of computing platform running the target tracking algorithm can provide benefits in tracking performance.

Both of these components, UAV and computing platform resource management pose difficult problems in and of themselves. Although the end goal is to apply NBO for joint control of UAV fleet and the computing platform, we first take a two prong approach: extending our UAV control problem to include real-world concerns (e.g. target ambiguity), Section 3, *and* formulating a computational resource management problem, Section 4. The final destination of this course of action is to unite the two branches to develop a more encompassing resource management approach.

2 Initial Work

Our initial work addressed both a framework for designing a planning and coordination algorithm for UAV fleet control in a target tracking setting and its implementation on GPU platforms for increased computational speed. The aim of this combination is to produce a viable real-time online control algorithm that copes with the basic complexities of the dynamic scenario. Specifically, the algorithm collects measurements generated by sensors on-board the UAVs, constructs tracks from those measurements and plans the future motion of the UAVs to

minimize tracking error in terms of mean squared error (MSE). The following features influenced the design of the planning algorithm.

Dynamic UAV motion constraints. The UAVs fly at a constant speed with bounded lateral acceleration, limiting the turning radii. The presence of dynamic constraints implies that the planning algorithm must include lookahead to achieve prime long-term performance.

Randomness. The measurements have random errors, and the models of target motion are random.

Spatially varying measurement error. The range error of the sensor is an affine function of the distance between the sensor and the target. The bearing error of the sensor is constant, translating to a proportional error in Cartesian space. This spatially varying error contributes to the richness of the sensor placement problem.

Tracking objectives. The performance objectives considered are defined as minimizing the MSE between tracks and targets.

2.1 Algorithmic Approach

The primary tracking objective within the UAV scenario is to maintain a position/velocity estimate of dynamic moving targets. This necessitates the use of a non-myopic formulation and solution approach, typically based on a POMDP framework. Since the POMDP formulation implemented for our investigation is the same as [10] we provide only an abridged description of the POMDP. Following this, we present a general introduction to the solution methodology also utilized in [10].

State. The state includes the sensors, target and tracker. This is represented by the tuple $x_k = (s_k, \zeta_k, \xi_k, P_k)$, respectively, where ξ_k as the mean and P_k as the covariance of the tracker.

Action. The formulation assumes constant velocity UAVs whose control inputs are lateral acceleration components. These inputs a_k essentially position the sensors.

State-transition law. The sensor state evolves according to the acceleration control described above. The target state evolves according to $\zeta_{k+1} = f(\zeta_k) + v_k$ where v_k represents an i.i.d. white Gaussian noise sequence and f represents a linear motion model. Finally, the track state (ξ_k, P_k) evolves according to a tracking algorithm that is defined by a data association method and Kalman filter update equations.

Observations and observation law. The sensor and track states are fully observable, so their observations are equivalent to their state components. The target state is unobservable; we only collect random measurements of the target state, which are functions of the locations of the sensors *and* targets. We receive a two-dimensional observation of range and azimuth defined as $z_k^{\zeta} = g(\zeta_k, s_k) + w_k$ where g gives the polar coordinates of the target with the sensor at the origin, and $w_k \sim \mathcal{N}(0, R(\zeta_k, s_k))$ with $R(\zeta_k, s_k)$ as the two-dimensional covariance matrix.

Cost function. In our tracking scenario we consider the mean squared error (MSE) defined by

$$C(x_k, a_k) = \underset{v_k, w_{k+1}}{\mathrm{E}} \left[\|\zeta_{k+1} - \xi_{k+1}\|^2 \mid x_k, a_k \right].$$

The above definitions completely characterize the POMDP framework on which our solution method is based, but after formulating a POMDP it can be converted into a belief-state MDP by considering the distribution of the unobservable state as the state component for the the MDP (the belief-state). In this case, we define the belief-state associated with the POMDP as $b_k(x_k) = P_{x_k}\left(x_k \mid z_0, \cdots, z_k, a_0, \cdots, a_{k-1}\right)$, which in turn is determined by the four components b_k^s, b_k^{ξ}, b_k^{ζ}, and b_k^P, where b_k^{ξ} is the distribution arising from the unobservable state

component, updated using Bayes Rule. This update can be implemented via an (extended) Kalman filter under the assumption of the proper tracking model, Gaussian statistics, and correct data association, yielding $b_k^{\xi} \sim \mathcal{N}(\xi_k, P_k)$. The cost function can then be rewritten in terms of the belief-state as $c(b_k, a_k) = \int \mathrm{E}_{v_k, w_{k+1}} \left[\|\zeta_{k+1} - \xi_{k+1}\|^2 \mid s_k, \zeta, \xi_k, a_k \right] b_k^{\zeta}(\zeta)\, d\zeta = \mathrm{Tr}\, P_{k+1}$, where Tr represents trace.

In the POMDP formulation, exact optimal solutions are practically infeasible to compute. Recent efforts have focused on obtaining *approximate* solutions (e.g., see [3, 4, 8]). We now briefly present one solution method, *nominal belief-state optimization* (NBO), described in detail in [10]. NBO is a receding-horizon method in which the effect of taking an action at the current belief state is approximated by a "nominal belief-state" trajectory $\hat{b}_k$ produced by substituting a nominal value of noise (zero) into the state transition law. The resulting approximate cumulative cost-to-go is then given by $J^*(b) \approx \min_{(a_k)_k} \sum_k c\left(\hat{b}_k, a_k\right)$ where $(a_k)_k$ is the ordered list of actions at each epoch $(a_0, a_1, \ldots)$.

2.2 GPU Parallelization

The algorithmic techniques based on NBO enable the resolution of aspects of the UAV tracking problem heretofore beyond the ability of existing techniques. Despite this, the algorithms are extremely computationally challenging. Experience with Matlab prototype codes suggests that even the simplistic scenarios described in [9] take an unacceptably long time to evaluate. For the scenarios studied in this paper, even anticipating that rewriting the Matlab codes in `C++` could provide speedups by over an order of magnitude, the execution time would not be sufficient for real-time performance.

In the initial application context, we considered GPUs to achieve real-time performance in the scenarios of interest. GPUs represent a viable trade-off between the performance of dedicated hardware such as field programmable gate arrays (FPGAs) and the programmability of general-purpose instruction-set processors.The parallelization techniques that we use are based on the *polyhedral model:* a single unifying mathematical foundation.

The overall approach consists of isolating performance-critical parts of the program as tight kernels with a precise, polyhedral structure. Such kernels correspond to program fragments that can be statically analyzed using tools that allow for a quantitative prediction of parallelism, schedule, and resource utilization, which includes both processing units, memory, and register resources. Through systematic *design space exploration* we can obtain a set of Pareto optimal parallel solutions for the target platform under consideration.

2.3 Summary of Methods and Results

The proof-of-concept was a Matlab simulation. Preliminary simulations indicated that this base algorithm substantially reduces the tracking error, compared to competing algorithms [10]. However, for sufficiently large systems of practical interest, the computational burden precludes real-time implementations of the method. We first identified the performance bottlenecks or "hot spots" that prevent its use in real-world scenarios.

The Matlab prototype code develops a flight plan (hereinafter called the acceleration plan) for each of the UAVs in the scenario given the current the target state estimate and the sensor state. This main function is called `GetAccelPlan` and determines the optimal acceleration plan by solving an optimization problem. Initially `GetAccelPlan` used Matlab's generic nonlinear optimization function, `Fmincon`, which itself repeatedly evaluates the objective function $c(b_k, a_k)$, which is also written in Matlab and passed to `Fmincon`. The objective function requires the

evaluation of a Kalman filter over a given time horizon, and because of the structure of the code (dense matrix algebra) it is an immediate candidate for GPU acceleration. In addition, the optimization algorithm `GetAccelPlan` is itself an important candidate.

Profiling the Matlab prototype code using a set of user-defined scenarios confirmed the initial hypotheses and also revealed a potential problem. Almost 70% of the execution time was indeed spent in the objective function but there were many thousands of independent calls to the function. Therefore, simple parallelization of just this function was unlikely to provide the desired gains—it would most likely be dominated by the latency of data transfer to the GPU, and would lack adequate work complexity and thus suffer from Amdahl's Law bottleneck. It would be necessary to develop a more coarse-grain parallelization where `GetAccelPlan` itself could be parallelized. Nevertheless, the gains of parallelization of the objective function calculation would be a critical component of the overall strategy. It also implied a dependence between the algorithm development and the parallelization tasks. This issue is resolved by making the design decision that any coarse grain parallelization would use multiple *independent* calls to the evaluation of the objective function.

2.4 Algorithmic Exploration

Our algorithmic exploration needed to satisfy two important constraints. First, because `Fmincon` is a proprietary, "black-box" library function provided by Matlab, we implemented an alternative nonlinear optimization function exploiting multiple simultaneous calls to evaluate the objective function. Furthermore, being cognizant of the single precision constraints of the target GPU, we also explored the precision behavior of the algorithms. Under single precision restrictions, certain limitations in the optimization algorithms became apparent.

We chose Particle Swarm Optimization (PSO) [7] for the function minimization required in `GetAccelPlan`. The use of PSO was motivated by the relatively simple implementation and the point evaluation driven minimization for course grain parallelism. PSO is a random search initialized by spreading a set of particles (points evaluated) over the domain, with inter-particle communication across the set motivating particle movement, the particles are drawn toward the minima of the function. The relative performance of `Fmincon` and PSO can be seen in Figure 1 with average tracking errors of 4.01m and 3.61m, respectively, in the two UAV, one target scenario. PSO was tested on various other scenarios showing equivalent relative performance.

Precision & Numerical Issues. While the original Matlab prototype began in completely double precision format the target GPU platform (NVIDIA Fermi series) offered significantly higher performance via single precision arithmetic. Discrepancies became apparent during the comparison of single precision `C` code (the basis for parallelization) and the Matlab version. These differeing results were traced to rounding errors caused by the inherent truncation.

`GetAccelPlan` is forced to optimize an objective function strictly returning single precision values. This decrease of precision in the function space caused difficulties in finding optima for `Fmincon`, but the single precision computations had no discerning effect on PSO. The comparison from double to single precision performance can be seen in Figure 1. The average tracking errors corresponding to `Fmincon` and PSO are 41.39m and 3.46m, respectively.

2.5 GPU Acceleration

The tight interdependence between the algorithmic development and GPU parallelization led to a two-stage parallelization effort consisting of a fine-grain component that would focus on the objective function and a coarse-grain parallelization of the entire optimization routine. This

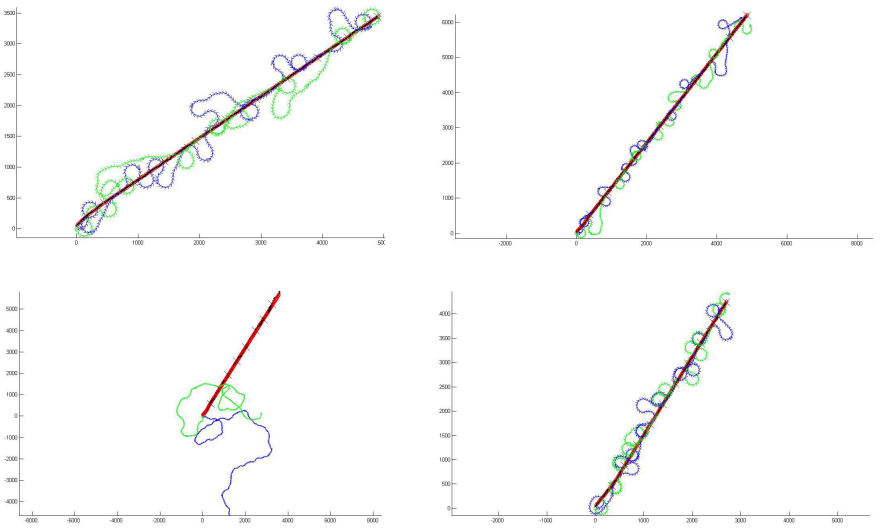

Figure 1: UAV trajectories illustrating the tracking error of double precision, `Fmincon` (4.01m) and PSO (3.61m) in the top row vs single precision `Fmincon` (41.39m) and PSO (3.46m) in the bottom row.

decoupled software development efforts. With the two-phase parallelization strategy identified, the objective function was the initial focus of GPU acceleration. Because the matrix sizes and time horizons are relatively small it is necessary to have multiple *independent* evaluations of this kernel to fully exploit GPU computational resources (coarse-grain parallelization), e.g., PSO. This led to an interesting design-space exploration.We first ported the objective function from Matlab to `C`, and then parallelized it in CUDA for the GPU. We faced critical challenges: (i) the dense matrix algebra required complicated and non-standard parallelization; (ii) the relatively small matrix sizes involved in the Kalman filter necessitated careful thread management.

When described using the polyhedral model, the computation was expressed in a 6-dimensional space. Three dimensions were involved in the iterations describing the matrix operations performed during the Kalman filter updates. Three additional dimensions corresponded to outer loops over (i) the targets, (ii) the sensors, and (iii) the time steps involved in the stochastic dynamic programming. Since the decision was taken to *not* parallelize the "inner" three dimensions, and there was an apparent strict dependence in the time dimension, it would seem that only two dimensions of parallelism could be exploited. However, there was a significant preprocessing step, namely the computation of the covariances for each of the targets for each sensor at each time step, that had three independent dimensions of parallelism. We therefore chose to perform a GPU parallelization using 3D threadblocks.

Once the high-level parallelization was identified, we optimized the code by systematically exploring trade-offs between parallelism and memory (at all levels of the GPU memory hierarchy—registers, shared memory, and global memory). We optimized global memory accesses through coalescing. Our final GPU parallelization of the objective function on an NVIDIA GTX-465 used 512 threads per threadblock, arranged in an $8 \times 8 \times 8$ cube.

The preliminary test results were very encouraging. We tested the GPU implementation with 264 independent instances of of the objective function calculation with 8 sensors, 8 targets,

for an 8-second time horizon, on an NVIDIA GTX-465, a card that has 11 vector-processors operating at 607 MHz. The total execution time, including host-GPU data transfer, was 4.07×10^{-3} sec. For comparison, the Matlab code, running on a core2, quad-core Intel Q8200 with 8GB memory under Windows took 10.8 sec (on a single core).

3 UAV Scenario Advancements

Our initial experiences with the parallelization and porting of the prototype control code to the GPU platform provided a basis for our current work. As mentioned in Sections 1 and 2, the effort to convert the original Matlab code to CUDA and optimize it for the GTX-465 was not trivial, and required heavy interaction between the algorithm and HPC components, resulting in an extensive number of man-hours expended. Thus, to be able to explore the plethora of platforms available, we engage the equational programming language of `Alpha` and its accompanying code generators. `Alpha` provides automatic compilation capabilities required for the aforementioned platform exploration/advancement. Since `Alpha` only deals with programs fitting the polyhedral model and `AlphaZ` is strictly capable of regular affine transformations, extensions are required to encompass the the algorithmic needs.

3.1 Scenario Formulation Extensions

Several modifications are slated for the extension of our UAV scenario. Each alteration incurs POMDP model refitting, NBO adaptations, and target tracking algorithm conversions. Perhaps one of the most needed augmentations to the UAV work from Section 2 is data association. The work in Section 2 made a dubious assumption: the origin (target) of the measurement is known. While this assumption can be valid in a limited number of cases (e.g. extreme separation of targets, highly discernible target features, etc.), this is not a common assumption for general target tracking. In fact, *data association* is a well studied complex problem with many relevant and available solution techniques.While we implement a prominent data association technique, multi-hypothesis tracking (MHT), our development lies in accommodating its addition in NBO. We extend the objective of the NBO beyond minimizing MSE to additionally incorporate the impact of the action selection on target ambiguity.

Under MHT's very general assumptions it accounts for missed detections, false alarms, track initiations (new target presence), and track deletions (target disappearance). Since the inception of MHT by Ried [14], numerous variations of the algorithm have been presented, from its original hypothesis oriented MHT (HOMHT), to the oft adopted track oriented MHT (TOMHT) [1]. The reduced computational loads and memory requirements offered by TOMHT make it a viable choice for our needs. Due to the ample available literature we do not present the MHT algorithm herein, but detail the POMDP and NBO enhancements inspired by its use.

Before proceeding to the definitions and extensions we clarify that the following are only the select components from Section 2 which are effected by the addition of the MHT data association algorithm. These descriptions include both POMDP components and modifications to the NBO algorithm motivated by the reduction of target ambiguity. This reduction is achieved not through MHT but by way of UAV control (NBO) as the capabilities can be extended to reduce target and observation ambiguity by actively controlling the incoming observation types and accounting for their effects in the action selection process.

3.1.1 POMDP Model Adjustments

The POMDP components directly affected by the addition of the MHT algorithm are the state and state transition law, the observation law, the cost and the belief state representation. The details of the changes to the components are listed below.

State and State-transition law. The state includes three subsystems: the sensors, the targets, and the track state. Both the sensor state s_k and the target state ζ_k remain unchanged. The incorporation of MHT does however effect the tracker state. The previous definitions simply represented the tracker state with the posterior mean vector ξ_k and the posterior covariance matrix P_k, standard in Kalman filtering algorithms. Using MHT combine with Kalman filtering provides a more complex representation of the tracker state as a Gaussian Mixture (GM), with sets of posterior means $\{\xi_k^i\}_i$, posterior covariances $\{P_k^i\}_i$ and corresponding weights $\{\omega_k^i\}_i$. The track state then transitions according to the Kalman filter MHT algorithm.

Observations and observation law. As in the state definition the four separate components remain the same and the observations for the fully observable components are equal to their true state. The observation of the state is similar in that it is the two-dimensional observation of the range and azimuth, z_k^ζ. However, MHT relaxes the assumptions to include false alarms, missed detections, new target arrivals and target deletions. Each of these additions is modeled with a corresponding probability of occurrence.

Cost function. The cost function acquires an additional term to include the minimization of target ambiguity. We form a linear combination of the MSE from our initial work and a statistical distance between respective target distributions. The statistical distance is the worst-case χ^2 distance. This is the same cost function and statistical distance presented in [11]. It should be noted that [11] shows the advantages of the worst-case χ^2 distance over other well known statistical distances. Even though this formulation is not wholly original it has yet to be implemented with MHT or combined with our proposed parallelization techniques.

Belief state. The belief state at time k is a *distribution* over states, $b_k(x) = \mathrm{P}_{x_k}\big(x \mid z_0, \ldots, z_k;\ a_0, \ldots, a_{k-1}\big)$. The belief state is given by $b_k = (b_k^s, b_k^\zeta, b_k^\xi, b_k^P)$ where only b_k^ζ is unobservable and is updated with z_k^ζ using Bayes theorem. The specifics of this Bayesian update are defined by the Kalman filter MHT.

3.1.2 NBO Modifications

The purpose of NBO is to select optimized action sequences by costing-out over a "nominal" progression of the belief state over a given time horizon. In our previous work this nominal progression was based on the posterior mean provided by the Kalman filter. Similarly in [11] the joint probability data association algorithm provided a single definitive mean per target. However, using MHT has provided us with an innovation opportunity. The belief state in our formulation is represented by a GM at each time step opposed to a single mean. We are currently studying three options for the nominal belief state propagation: (i) deriving a single mean by computing the weighted average of the GM, (ii) propagating the most likely component of the original GM, and (iii) treating each component of the GM as an independent description of the belief state itself and propitiating them separately and selecting an action for each, in turn combining the action selections either by averaging or majority votes. We are exploring these options two fold, both analytically and empirically, for the effect on our parallelization strategy.

3.2 Alpha Developments

The PSO code is an algorithm which searches for the best candidate solution by letting a group of candidate solutions evolve in the search space. At every iteration, each candidate position is updated according to the best local (known by this specific candidate) and the global best solution (known by all particles). The termination condition for PSO is unknown statically, making the number of iterations required to achieve this criteria unknown during code generation. Some operations, in particular the update of the velocity must respect the constraints, requiring another extension beyond the classical polyhedral model. The improvements to `Alphaz` required to generate code for PSO algorithm include the ability to run (potentially) unbounded iterative programs and the the use of dynamic dependencies inside a polyhedral program. These improvements described in [12].

Code Snippet. Listing 1 shows the two extensions in an Alphabets code snippet. Line 2 shows a while loop (over `t`) with a condition defined inside the C function `cond(int t)`. Line 5 and line 13 show the the usage of a variable defined after code generation. Depending on the value of `Z` at the point `i` (either 1 or 2) the value of `Y[i,t]` will decreased or increased.

Listing 1: While Loop and Dynamic Dependencies

```
1  boolean cond(t); //cond will be evaluated at every iteration
   affine counter {N|N>2} over {t|t>0} while cond(t)
   inputs
           float Init{i|0<i<N};
           dep Z {i->j|0<i<N && 1<=j<=2};
6  outputs
           float Y {i|0<i<N};
   lets
           Y[i,t] = case
           {|t==0}: Init[i,0];
11         {|t<=i}: Y[i,0];
           {|t>i && i==0}: Y[i,0];
           {|t>i && i>0 && Z[i]==2}: Y[i,t-i]-1;
           {|t>i && i>0 && Z[i]==1}: Y[i,t-i]+1;
           esac;
```

`AlphaZ` must determine the size of the memory for each variable when generating the code. For example, the `Init` uses, by default, an array of size `N`. Since we now allow while loop we must ensure that every `Alphabets` program uses strictly bounded memory quantities.

Finite memory allocation. By analyzing the type of dependencies within the program, we determine an upper bound on the amount of memory the program requires. When the dependence vector is uniform (the point `(i,t)` depends on `(i,t+a)`, `a` being some constant) the number of slices that may need to be saved is `a`. The size of the memory needed for the program is determined by taking the maximum of all the dependencies. However if non-uniform dependencies exist, it can be shown that the amount of slices needed is still bounded. In Listing 1, `i` used in the second part of `Y[i,t-i]` at lines 12 and 13 is inside the interval $[0, \min(t, N)]$. This implies at most `N` slices are needed.

Speculation. If the while loop condition is stationary (i.e. if the condition is false for some `t`, then it is false for all superior `t`), the condition test need not be executed for every iteration and only executed every X iterations. A fallback/rewind mechanism is implemented to find the *exact* smallest `t` where the condition is initially false.

4 Dynamically Changing Computational Resources

We now describe the POMDP formulation allowing for the problem of retaining computational efficacy of a long-running program in the face of dynamically changing resources as are expected to occur in a DDDAS scenario. We also describe our ongoing efforts in extending polyhedral compilation and parallelization tools and the `AlphaZ` system to resolve these problems.

Assume that we seek an optimal implementation of a long-running polyhedral program. In the context of the UAV control, the control algorithm to optimize the tracking objective is, in fact such a program. It needs to be optimally parallelized on an embedded platform, possibly under very tight real-time constraints, and limits on computation resources, as well as on energy consumption. The goal of this scenario is to optimize a computational objective function, representing execution time and energy. Under some further simplifying assumptions, energy can be directly derived from execution time.

Furthermore, because of the dynamic nature of the environment, computing resources such as the number of cores available and the amount of cache memory available may change. This is modeled by the stochastic "external events," such as the arrival of a competing task, a rise in the ambient temperature or a failure of a core. Each such event is characterized by its anticipated time and duration (i.e. Poisson process), and the factors by which it modifies the hardware compute resources.

Since the executing program fits the polyhedral model, we assume that analytic measures of execution time and energy consumption of the code are available. These include both tunable parameters (e.g., number of threads) and problem specific parameters (e.g., data size, core count, etc). Existing literature provides us methods to optimally choose the tunable parameters for any specific value of the problem/machine parameters. These methods range from specialized closed-form expressions to empirical searches through a space of candidate solutions. In our scenario we assume the former, in the interest of simplicity. Once the new values of the code parameters are known however, the executing program must be "locally interrupted," its parameters changed and then execution must resume. We assume that this step involves an overhead cost.

Now, a direct and immediate response to an external event of changing the tunable parameters, may reduce the expected completion time, but the overhead cost for doing so may become unnecessarily large if the external event is short lived. Thus, such an immediate response will be myopic, and the theory of POMDPs becomes relevant. Our ongoing work involves refining this scenario to more realistic cases and then formulating and solving the resulting control problems.

4.1 Automatic Tiling and Code generation

A polyhedral compilation infrastructure needs to be extended in two primary directions: (i) generation of maximally parametric code, and (ii) late binding parameters.

Maximally parametric code refers to a code generation strategy where as many choices as possible are left as parameters. This includes but is not limited to: (i) number of threads, (ii) number of tiling levels, (iii) the objective (locality or parallelism) of each tiling level,(iv) the tile sizes at each level, (v) the tile shapes at each level, (vi) the schedules between (vii) the schedules within tiles, and (viii) the memory allocation of data/iterations. Most polyhedral compilation strategies require fixed size tiles and a fixed number of levels.

Late binding refers to techniques by which we defer to run-time the specific choice of all the parameters that have not been specified at compile time. As mentioned above, this is done by resolving as many of the optimization problems at compile time in closed form so that only these formulas need be evaluated at run-time.

In addition to this, the code generators need to be modified so that the code can be "locally interrupted" through a mechanism similar to checkpointing. Changing the tile size dynamically may also require a data-redistribution step.Such a redistribution is not necessarily an affine transformation and it must be included in the cod generation.

We have developed Constant Aspect Ratio Tiling (CART) which is a technique that takes an `Alpha` program and automatically tiles the program. For a given program, CART tiles the whole computation space into rectangles, and the tile sizes along different dimensions are constant multiples of the same tile size parameter. The constant multiples for all the dimensions are called *Constant Aspect Ratios*, which are given as inputs to CART. Then a code generator produces the corresponding C+OpenMP code. Other than the original program inputs, the generated code also takes one tile size parameter as input. All the techniques are integrated in a polyhedral framework called `AlphaZ`.

5 Conclusion and Future Work

The general resource management problem spans several areas of interest. The two specific areas focused on here are sensor resource management and computational resource management. The investigation of the sensor resource management in Section 2 provided viable performance improvements in both tracking error and execution time. However, this study also highlighted lacking elements for real world implementation. These elements are a current focus of our research and are discussed in Section 3. Lastly, the overall research with sensor resource management revealed the scope of our control algorithm could be increased to include, not only the sensor system, but also the computational platform executing the tracking algorithms. Through the non-myopic management of the computational resources, Section 4, energy and time expended for the tracking execution can be reduced in an optimal manner. This, in turn, may yield better tracking performance by allowing the processing of additional valuable sensor and system data.

Section 4 is a precursory overview of our approach, it provides an avenue for future work. This work consists of continued formulation and algorithm development. Both of these steps contain open problems of interest. The POMDP formulation requires extensive research to compose accurate models on dynamically changing voltage, core usage, and cache memory. Also, our approach for automatic code generation requires considerations for scheduling and memory mapping changes if the tile shape should be changed. Finally, we are considering the inclusion of failure resilience without the standard usage of check-pointing.

References

[1] S. S. Blackman. Multiple hypothesis tracking for multiple target tracking. *Aerospace and Electronic Systems Magazine, IEEE*, 19(1):5–18, Jan 2004.

[2] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. Pluto: A practical and fully automatic polyhedral program optimization system. In *ACM Conference on Programming Language Design and Implementation*, pages 101–113, Tuscon, AZ, June 2008. ACM SIGPLAN.

[3] E. K. P. Chong, C. Kreucher, and A. O. Hero. POMDP approximation methods based on heuristics and simulation. In A. O. Hero, D. Castañon, D. Cochran, and K. Kastella, editors, *Foundations and Applications of Sensor Management*, chapter 8, pages 95–120. Springer, 2008.

[4] E. K. P. Chong, C. Kreucher, and A. O. Hero. Partially observable Markov decision process approximations for adaptive sensing. *Discrete Event Dynamic Systems*, 19:377–422, 2009. 10.1007/s10626-009-0071-x.

[5] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[6] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part I. One-dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, October 1992.

[7] J. Kennedy, R. Eberhart, et al. Particle swarm optimization. In *Proceedings of IEEE international conference on neural networks*, volume 4, pages 1942–1948. Perth, Australia, 1995.

[8] Y. Li, L. W. Krakow, E. K. P. Chong, and K. N. Groom. Approximate stochastic dynamic programming for sensor scheduling to track multiple targets. *Digital Signal Processing*, 19(6):978 – 989, 2009. DASP'06 - Defense Applications of Signal Processing.

[9] S. Miller, Z. Harris, and E. K. P. Chong. A pomdp framework for coordinated guidance of autonomous UAVs for multitarget tracking. *EURASIP Journal on Advances in Signal Processing*, 2009(1):724597, 2009.

[10] S. A. Miller, Z. A. Harris, and E. K. P. Chong. A POMDP framework for coordinated guidance of autonomous uavs for multitarget tracking. *EURASIP J. Adv. Signal Process*, 2009:2:1–2:17, January 2009.

[11] S. Ragi and E. K. P. Chong. UAV path planning in a dynamic environment via partially observable Markov decision process. *Aerospace and Electronic Systems, IEEE Transactions on*, 49(4):2397–2412, OCTOBER 2013.

[12] S. V. Rajopadhye, S. Gupta, and D. Kim. Alphabets: An extended polyhedral equational language. In *IPDPS Workshops*, pages 656–664. IEEE, 2011.

[13] S. V. Rajopadhye, S. Purushothaman, and R. M. Fujimoto. On synthesizing systolic arrays from recurrence equations with linear dependencies. In *Proc. of the sixth conference on Foundations of software technology and theoretical computer science*, pages 488–503, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[14] D. B. Reid. An algorithm for tracking multiple targets. *Automatic Control, IEEE Transactions on*, 24(6):843–854, Dec 1979.

[15] T. Yuki, G. Gupta, D. Kim, T. Pathan, and S. V. Rajopadhye. Alphaz: A system for design space exploration in the polyhedral model. In H. Kasahara and K. Kimura, editors, *LCPC*, volume 7760 of *Lecture Notes in Computer Science*, pages 17–31. Springer, 2012.