AN EXPLORATION OF PARALLELISM FOR SOLVING MULTIPLE MATH PROGRAMMING INSTANCES USING A COMMERCIAL SOLVER

Carson G. Long, Department of Operational Sciences, Air Force Institute of Technology, 2950 Hobson Way, Wright-Patterson AFB, OH 45433, 937-255-6565, carsonglong@gmail.com

Brian J. Lunday, Department of Operational Sciences, Air Force Institute of Technology, 2950 Hobson Way, Wright-Patterson AFB, OH 45433, 937-255-6565 x4624, brian.lunday@afit.edu

ABSTRACT

This paper introduces a problem that motivates the use of parallelism when invoking commercial solvers. It formulates the network vulnerability identification problem as a bilevel programming problem. Although a customized genetic algorithm (GA) is an effective metaheuristic for this $\mathcal{NP}$-hard problem, it is not typically efficient because it assesses population member fitness functions sequentially. Moreover, our problem's fitness function is separable over the shipments being routed. This work documents several endeavors to parallelize this latter aspect of optimal shipment routing, ultimately identifying an opportunity with specific recommendations for software developers to advance the state of optimization software and computing capability.

Keywords: Mathematical Programming, Parallelism, Commercial Optimization Solver

INTRODUCTION

Advances in technology and computing have enabled more powerful computing resources, allowing analysis efforts to examine and solve larger problems with increasing levels of computational complexity. A component of this improved capability is the distribution of computational expense across resources to reduce overall execution time of processing tasks. Herein, we explore approaches to parallelism that may be implemented within a bilevel modeling framework to more efficiently solve the underlying problem.

Motivating and supporting our research, the United States Transportation Command is a functional commands that supports warfighting commands, military services, and defense agencies. As a component of its globally integrated mobility operations, the command provides intratheater movement of material and assets via aerial and seaborne platforms to respective air and sea ports of debarkation in an area of operations (AOR). Once material and assets reach an AOR, they must traverse a ground distribution network (e.g., railway, roads) to reach commanders who need them. Each shipment has both preferred and required deadlines for delivery. Shipments delivered after required deadlines are considered to be too late to provide practical value for the commander.

A ground distribution network within an AOR is vulnerable to disruptions from natural events like severe weather, benign events like refugee evacuations, and malign events like deliberate attacks on network infrastructure. Such disruptions may temporarily prevent movement along portions of the network, preventing its use until one can remove obstructions and/or affect repairs.Such events represent vulnerabilities to material distribution, and the most impactful disruptions indicate the greatest vulnerabilities of concern. It is important to identifying these vulnerabilities, both in terms of their timing and location, so decision makers can take preemptive action to protect against them.

This research sets forth a bilevel mathematical programming model and accompanying metaheuristic solution method to identify spatiotemporal vulnerabilities for a ground distribution network, given a decision-maker seeking to effectively route material. The bilevel program is representative of an attacker-defender, Stackelberg game theoretic framework, wherein an intelligent attacker with a fixed number of limited-duration attacks disrupt a subset of arcs within a ground distribution network, and a defender with full knowledge of the attacks optimally routes their shipments.

To solve instances of our model, this research uses a customized genetic algorithm to search the upper-level attacker's feasible region for high quality solutions. The quality of each solution of attacker interdictions is determined by solving the lower-level defender's material routing problem, i.e., identifying a defender's best response. To find each such solution to the lower-level problem, one must invoke a commercial optimization solver, a computationally expensive procedure. This computational burden motivates our exploration of alternative parallel processing procedures to invoke a leading commercial solver to improve the computational efficiency of the underlying genetic algorithm.

BILEVEL MATH PROGRAMMING FORMULATION

It is relevant to understand both the math program and the accompanying metaheuristic. This section presents the bilevel mathematical programming formulation. Within it, an upper-level attacker seeks to identify a fixed number of limited-duration attacks to maximize a single objective. In response to and with full knowledge of the disruptive attacks, a lower-level defender routes multiple shipments between respective origin-destination pairs while minimizing multiple objective functions.

Prior to presenting the formulation, it is first necessary to define the following, sets, parameters, and decision variables.

Sets

- $K = \{1, ..., \mathcal{K}\}$: The set of shipments, indexed by k
- $N = \{1, ..., \mathcal{N}\}$: The set of nodes within the transportation network, indexed by i or j. For each shipment k, i_{s_k} and i_{d_k} respectively indicate its origin and destination nodes.
- A: The set of directed arcs in the network indexed by (i, j)

- $G(N, A)$: The complete network composed of nodes N and arcs A
- Π : The set of attacks conducted by an adversary, indexed by π, each of which is specific to a location (i.e., an arc) and a time
- $R(\Pi)$: The rational reaction set of alternative optimal solutions for the lower-level problem, given Π

Parameters

- v_k: the volume of shipment k
- b_{ik}: a parameter equal to 1 if shipment k originates at node i; -1 if shipment k must be supplied to node i; and 0 otherwise
- l_{ij}: The length of arc (i, j)
- τ_{ij}: The time required to traverse arc (i, j)
- $\hat{t}_k$: The earliest time at which shipment k is available to depart node i_{s_k}
- t_k^{EA}: The earliest time at which shipment k *should* arrive at node i_{d_k} without incurring an Early Arrival Penalty (EAP_k)
- $t_k^{LA} \in T$: The latest time as which shipment k *should* arrive at node i_{d_k} without incurring a Late Arrival Penalty (LAP_k)
- $t_k^{RA} \in T$: The required arrival time of shipment k to node $i_{d_k} \in N$. Whereas t_k^{LA} can be violated with a penalty, t_k^{RA} cannot be violated.
- δ : The duration of an adversary attack. All attacks are assumed to have a homogeneous duration.

Decision Variables

- $(i, j)_\pi$: The arc (i, j) disrupted by adversary attack π
- α_π : The time at which adversary attack π begins against arc $(i, j)_\pi$
- x_{ijk}: A binary decision variable equal to 1 if shipment k traverses arc (i, j), 0 otherwise
- t_{ik}: The time at which shipment k arrives at node i, if shipment k traverses the node; 0 otherwise
- EAP_k: The penalty associated with receiving demand of shipment k prior to the earliest arrival time (t_k^{EA}) at node i_{d_k}
- LAP_k: The penalty associated with receiving demand of shipment k after the latest arrival time (t_k^{LA}) at node i_{d_k}

- $y_{k\pi}$: A binary decision variable equal to 1 if shipment k traverses arc $(i,j)_\pi$ *after* an attack during duration $[\alpha_\pi, \alpha_\pi + \delta]$, and 0 if it traverses the arc *prior to* the attack. (A shipment cannot traverse an arc while it is being disrupted by an attack.)

Given the aforementioned notation, we formulate the **Bilevel Material Routing Problem** (**BMRP**) model as follows. The upper-level decision-maker seeks to solve the following problem:

$$\max_{\pi \in \Pi} \; f_1(\boldsymbol{x}) \tag{1a}$$
$$\text{s.t. } (i,j)_\pi \in A, \; \forall \; \pi \in \Pi, \tag{1b}$$
$$\alpha_\pi \geq 0, \; \forall \; \pi \in \Pi, \tag{1c}$$
$$(\boldsymbol{x}, \boldsymbol{t}, \boldsymbol{EAP}, \boldsymbol{LAP}) \in R(\Pi). \tag{1d}$$

The attacker maximizes the shipment volume-weighted distance traversed by the lower-level decision-maker. For $|\Pi|$ attacks, Constraint (1b) limits each attack to an arc on the network, and Constraint (1c) bounds the earliest start time for each attack on a continuous time horizon). Constraint (1d) stipulates that a lower-level decision maker's actions $(\boldsymbol{x}, \boldsymbol{t}, \boldsymbol{EAP}, \boldsymbol{LAP})$ will belong to the reaction set $R(\Pi)$, i.e., the rational reaction set. For this formulation with multiple lower-level objective functions, $R(\Pi)$ is comprised of the set of Pareto optimal solutions for the lower-level problem, determined by solving the math program expressed via (2a)–(2p).

$$\min_{\substack{\boldsymbol{t},\boldsymbol{x},\boldsymbol{y} \\ \boldsymbol{EAP},\boldsymbol{LAP}}} \quad (f_1(\boldsymbol{x}), f_2(\boldsymbol{t}), f_3(\boldsymbol{EAP}, \boldsymbol{LAP})) \tag{2a}$$
$$\text{s.t. } f_1(\boldsymbol{x}) = \sum_{(i,j)\in A} \sum_{k \in K} l_{ij} v_k x_{ijk}, \tag{2b}$$
$$f_2(\boldsymbol{t}) = \sum_{k \in K} \left(t_{i_{d_k} k} - \hat{t}_k \right), \tag{2c}$$
$$f_3(\boldsymbol{EAP}, \boldsymbol{LAP}) = \sum_{k \in K} \left(EAP_k + LAP_k \right), \tag{2d}$$
$$\sum_{j:(i,j)\in A} x_{ijk} - \sum_{j:(j,i)\in A} x_{jik} = b_{ik}, \; \forall \; i \in N, k \in K, \tag{2e}$$
$$t_{i_{s_k} k} = \hat{t}_k, \; \forall \; k \in K, \tag{2f}$$
$$t_{jk} \geq t_{ik} + \tau_{ij} - M(1 - x_{ijk}), \; \forall \; (i,j) \in A, k \in K, \tag{2g}$$
$$EAP_k \geq v_k \left(t_k^{EA} - t_{i_{d_k} k} \right), \; \forall \; k \in K, \tag{2h}$$
$$LAP_k \geq v_k \left(t_{i_{d_k} k} - t_k^{LA} \right), \; \forall \; k \in K, \tag{2i}$$
$$t_{i_{d_k}} \leq t_k^{RA}, \; \forall \; k \in K, \tag{2j}$$
$$t_{jk} - M(1 - x_{ijk}) \leq \alpha_\pi + M y_{k\pi}, \; \forall \; (i,j)_\pi, \; k \in K, \; \pi \in \Pi, \tag{2k}$$
$$t_{jk} - \tau_{ij} + M(1 - x_{ijk}) \geq (\alpha_\pi + \delta) y_{k\pi}, \; \forall \; (i,j)_\pi, \; k \in K, \; \pi \in \Pi, \tag{2l}$$
$$EAP_k, LAP_k \geq 0, \; \forall \; k \in K, \tag{2m}$$

$$x_{ijk} \in \{0,1\},\ \forall\ (i,j) \in A,\ k \in K, \tag{2n}$$
$$t_{ik} \geq 0,\ \forall\ i \in N,\ k \in K, \tag{2o}$$
$$y_{k\pi} \in \{0,1\},\ \forall\ k \in K,\ \pi \in \Pi. \tag{2p}$$

In the objective function (2a), the defender minimizes multiple objectives. The first objective (2b) computes for the total shipment volume-distance traveled. The second objective (2c) calculates the total transit time of material traversing the network. Of note, transit time measures the duration of time that shipments are moving on arcs as well as any loiter time en route to their destinations. The third objective (2d) accumulates any Early Arrival Penalties (EAP_k) and Late Arrival Penalties (LAP_k) for shipments arriving prior to or later than their desired delivery window.

Constraint (2e) enforces conservation of flow. Constraint (2f) indicates the time when each shipment k is available to depart node its origin, i_{s_k}. Constraint (2g) imposes lower bounds on the time at which each shipment k reaches every node j. If a shipment does not traverse an arc (i,j), this constraint effectively yields $t_{jk} \geq 0$; otherwise, the earliest it can arrive at node j is $(t_i + \tau_{ij})$. It suffices to set $M = \sum_{(i,j)\in A} \tau_{ij}$. Of note, these constraints do not impose a lower bound on a shipment's arrival when traversing between two nodes in the absence of interdiction on the connecting arc; a shipment may take more time to traverse the arc than τ_{ij} if conditions warrant (e.g., to wait for inclement weather or adversarial threat to dissipate), but it may never travel faster.

Constraints (2h) and (2i) respectively impose lower bounds on early arrival penalties (EAP_k) and late arrival penalties (LAP_k). Informing these bounds are both the amount of material in a shipment, v_k, and the number of days a shipment is either early or late, compared to the predefined delivery time window. This calculation leverages penalties linearly proportional to the magnitude of a shipment's earliness or lateness, but other alternatives are possible. Constraint (2j) enforces that shipments must reach their respective destination nodes i_{d_k} by their required arrival times.

Constraints (2k) and (2l) enforce three possible outcomes for each shipment and attack combination. First, a shipment may not traverse the arc ($x_{ijk} = 0$). Second, it may traverse the arc and reach node j no later than when when the attack begins, α_π. Finally, it may traverse the arc, departing node i no earlier than when the attack is complete, $(\alpha_\pi + \delta)$. If shipment k does not traverse arc $(i,j)_\pi$, any feasible $y_{k\pi}$-value satisfies both constraints. If a shipment k does traverse arc $(i,j)_\pi$ prior to the beginning of the attack (i.e., $y_{k\pi} = 0$), Constraint (2k) imposes an upper bound on t_{ik}. If it traverses the arc after the attack has concluded (i.e., $y_{k\pi} = 1$), Constraint (2l) imposes a lower bound on t_{jk}.

Constraint (2m) and Constraint (2n) enforces non-negativity and binary restrictions on associated decision variables.

Helpful to a solution procedure, the lower-level problem is separable by shipment $k \in K$. It may not be necessary to solve the problem separably for smaller instances, but it would be beneficial to do so for instances having a large number of shipments. This characteristic provides one motivation to explore parallel processing to invoke a commercial solver to address multiple instances of a math program, simultaneously.

CUSTOMIZED GENETIC ALGORITHM

The other motivation to explore parallel processing arises from the solution method we embrace to solve this problem. As previously described, we adopt a customized genetic algorithm to search the upper-level feasible region, seeking high-quality solutions for the attacker. For each such, population member, we invoke a commercial solver to identify the optimal routing response for the defender.

Algorithm 1 depicts the customized genetic algorithm. From generation to generation, the procedure iteratively performs crossover, mutation, and immigration operations on members of a population, thereafter attriting the least fit members to maintain a fixed population of size m. These operations are applied to pairs of members in the population, identified by their ordinal fitness function ranking.

Algorithm 1 Customized Genetic Algorithm

```
Initialize a population having m members (i.e., feasible attacker solutions)
while no termination criterion is met do
    Pair solutions based on ordinal ranking of fitness function values
    for all solution pairs (i.e., parents) do
        With probability p_c, apply crossover to generate two child solutions
        With probability p_m, mutate a single parent to generate a new solution
        With probability p_i, generate and immigrate a new solution
    end for
    Evaluate the fitness function of any newly created solutions
    Apply attrition to maintain population of size m
```

In preliminary testing using a population of $m = 10$ members on a realistic-sized instance, each fitness function evaluation required 5-10 minutes of computational effort with a leading commercial solver (i.e., Gurobi v9.5.1). Of importance, the crossover, mutation, and immigration operations are not sequence dependent, either within a category of operations or between them. In the rare occurrence that every possible operation applies to a single generation having m population members, there will be $1.5m$ new members to assess, prior to attrition. However, high values of p_c, p_m, and p_i are still unwieldy, in terms of imposing computational demands on the solution procedure.

EVALUATING PARALLELISM FOR MATH PROGRAMMING INSTANCES

As discussed, there are two potential benefits to parallel invocation of a commercial optimization solver. First, because this research is interested in routing a larger number of shipments (approximately 75-100), evaluating methods to implement parallelism is worth exploration since computation times to solve the lower-level problem might be reduced by a factor of the number of processors available if the routing of each shipment is solved in parallel. Second, Algorithm 1's serial evaluation of population member fitness functions could be implemented in parallel to save time.

With respect to implementation, this research implemented Algorithm 1 in Python while using the GurobiPy package to invoke Gurobi when solving instances of the lower-level problem (Gurobi Optimization, 2020). Both Python and Gurobi have built-in parallel processing packages that may

potentially be integrated into the lower-level routing solution routine or among the operations of crossover, mutation, and immigration in Lines 4-8 in Algorithm 1. Table 1 shows specific opportunities within Algorithm 1 wherein parallelism might be implemented. Testing evaluated the parallelism approaches for efficacy; the following sections discuss our findings.

Table 1: Parallelism Opportunities within Algorithm 1

		Methods Investigated	
Opportunity	**When**	**Gurobi**	**Python**
Evaluate a fitness function by routing $\mathcal{K}$	Line 1	Multiple scenarios	multiprocessing
shipments via separable math programs	Line 9	n/a	joblib, parfor
Perform GA operations on pairs of solutions	Lines 4-8	n/a	parfor

Parallelism Technique #1 - Multiple Scenarios in Gurobi

The multiple scenario approach arises from a Gurobi-specific Python package. GurobiPy integrates the Gurobi solver's capability to define multiple scenarios from a baseline model. Scenarios are generated from the baseline model by modifying (linear) objective function coefficients, variable lower/upper bounds, and constraint right hand side values (Gurobi Optimization, 2023). Once scenarios are defined, Gurobi solves them in parallel to return a set of solutions. In practical terms, this procedure is akin to Sample Average Approximation for a stochastic math program having some degree of parametric uncertainty.

With respect to Algorithm 1, when evaluating a member of a GA population by identifying the optimal routing of each shipment via a separate math program, there exists minor differences between the instances. From this perspective, there exists parametric variation among the $\mathcal{K}$ different lower-level problems that is arguably well suited for the multiple scenario approach.

Ideally, Gurobi would assign each scenario (i.e., solving for the routing solution of a single shipment) to a single available thread or core and return the optimal routing solution before moving to the next scenario in the queue until all shipments were optimally routed.

However, efforts to implement the multiple scenario approach revealed that the perceived parametric variations were too great. In practice, the resulting modifications to origins, destinations, and delivery windows across the different shipments departed from the types of variations GurobiPy can use for this solution technique. In testing, the solver reverted to solving the $\mathcal{K}$ separable lower-level problems in serial rather than parallel. As such, the multiple scenario feature is better suited for use when solving stochastic programs, e.g., via sample average approximation.

Parallelism Technique #2 - Multiprocessing in Python

The multiprocessing package in Python enables process-based parallelism to create individual environments for each process and to pool processes for execution by the computational resources available on a computer. The Python multi-processing module is an attractive alternative because it allows a developer to leverage multiple processors on a given machine (Heimes, 2023).

We made small adjustments to the coding structure of Algorithm 1 to test multiprocessing capability

within the bilevel modeling framework. When evaluating the fitness function of an upper-level solution, multiprocessing created $\mathcal{K}$ routing optimization problems for each lower-level shipment. That is, it simultaneously created $\mathcal{K}$ Gurobi models in their own environments.

Testing of this technique attained mixed results. Procedurally, it was successful. The multiprocessing Python package was capable of generating individual environments for processes and solving the shipments in parallel using available computing resources available. Computationally, it did not reduce the required time. The creation and disposal of environments added unanticipated computational expense that was non-trivial, compared to the time to optimally route single shipments. The result was an increase in the required computational effort to solve a lower-level problem instance.

Parallel computing using multiprocessing requires an acknowledgement of certain tradeoffs. Multiprocessing with Python requires overhead routines to distribute jobs from the pool to the processors. The expense of these routines was further encumbered by the need to repeatedly pass a large amount of instance-specific data (e.g., network topology data, temporal parameter data) to and from environments. Based on complementary testing of the multiprocessing function to ensure it worked, We identified both of these factors as obstacles to improve the computational efficiency for routing $\mathcal{K}$ shipments for our problem.

There are two possible measures to address this shortcoming. First, one could work to reduce the time required by overhead routines to allocate jobs to the processors. Second, one could seek to reduce the data requirements of an instance within each environment. Frankly, neither such approach is realistic for our problem. More realistic is to identify the types of optimization problems for which multiprocessing *would* be beneficial. Such problems should be difficult to solve but small, in terms of their parametric data. That is, $\mathcal{NP}$-hard combinatorial optimization problems should be well suited for this approach.

Parallelism Technique #3 - Joblib in Python

Next, testing evaluated the joblib Python library for its ability to enable parallelism. The joblib object uses "workers" to compute the application of a function to many different arguments in parallel (Varoquaux, 2023). Joblib contains 'Dump' and 'Load' operations to improve efficiency when processing of large datasets, specifically large numpy arrays.

With respect to Algorithm 1, we used joblib to route each of the $\mathcal{K}$ shipments when identifying an optimal routing solution for a new member of the population created via crossover, mutation, or immigration. Doing so entailed slight modifications to the coding of Algorithm 1 to add structure and management of the environments necessary to use the joblib package. We successfully implemented parallelism via the joblib package. However, its lack of thread management and inability to efficiently create, assign, and dispose of individual environments responsible for solving and returning the solution for each math program in parallel, exposed limitations of the joblib implementation.

Routing the shipments in parallel was much slower than solving the shipments in sequence. As with multiprocessing, the overhead tasks that assign jobs to workers are the primary suspect, requiring additional computational effort while not adequately managing the threads. To be succinct, overloaded threads do not leave adequate random access memory for both the centralized, over-

head tasks and the operating system to operate efficiently. The results are memory congestion and reducing processing speed, which in turn slow down the completion of tasks. Similar to the multiprocessing approach previously discussed, the joblib library could be improved by a combination of reducing overhead operations and implementing thread management, reserving a buffer of space for operations requiring a surge in memory. As with the multiprocessing technique, we also suspect the data-encumbered nature of these instances were also relatively challenging to the joblib package; it may be better suited for smaller yet computationally challenging optimization problems.

Parallelism Technique #4 - Parfor in Python

The final excursion integrated the parfor function, originally developed in Matlab and recently adapted for Python. Parfor decorates (i.e., prepends) an iterable function and returns the result of that function evaluated in parallel for each iteration over a specified range of values (Pomp, 2023).

We leveraged parfor in two different ways. First, we used it to route $\mathcal{K}$ shipments for a given upper-level solution (i.e., to evaluate a fitness function). Second, we used it to implement the loop represented by Lines 4-7 for each pair of ordinally ranked fitness functions.

When implementing parfor for the first use, it was able to leverage Gurobi to solve the shipments in parallel via function-based calls and utilize available computing resources. However, we again attained a computationally deficient result, relative to a serial implementation. The parfor function implementation was also not thread safe when leveraging Gurobi. Compared to simple command calls, a parallel creation and solving of individual Gurobi model instances is non-trivial.

When implementing parfor for the second use, we attained qualified success. Algorithm 1 examined each pair of solutions in parallel, conducting the respective GA operations. This implementation yielded marginal improvement in computation time, examining each ordinally-ranked pair of solutions in parallel and, for each one, conducting the specific GA operations in serial using Python dictionaries. The result was a more efficient generation of new solutions. Admittedly, this improvement affected an aspect of Algorithm 1 that did not use the commercial solver Gurobi, so the potential benefit was not as great as the other attempts to leverage parallelism.

CONCLUSIONS AND RECOMMENDATIONS

This research examined the use of parallel processing to speed up a metaheuristic designed to solve a bilevel programming problem. Although we used a genetic algorithm, similar metaheuristics are increasingly used by researchers to address bilevel programs, searching the upper-level feasible region and, for each solution, evaluating the solution by finding the lower-level decision maker's best response, i.e., solving an optimization problem. For a GA metaheuristic, the existence of a population of upper-level solutions that requires the solution to multiple lower-level problems within an iteration of the metaheuristic provides an opportunity to leverage parallel computing. Moreover, when a lower-level problem is separable as in our motivating problem, there is an additional opportunity to leverage parallelism by solving the separable programs simultaneously.

We identified and tested four techniques to leverage such parallelism, focusing primarily on using it

to solve the separable math programs when routing $\mathcal{K}$ shipments when evaluating a single upper-level feasible solution. For the scope of this study, we did not seek to parallelize the evaluation of multiple fitness functions simultaneously until we could successfully implement the former application.

Among the techniques examined were one specific to the GurobiPy package and three specific to the Python programming language. When invoking the Gurobi commercial solver, we tested the multiple scenarios feature within the GurobiPy package. Alternatively, we examined multiprocessing, the joblib library, and the parfor function to accomplish the same task at different points within the GA. Testing additionally leveraged parfor for parallel GA operations to create new population members. Each excursion demanded small alterations to the modeling framework coding structure to test integration of parallel computing.

Although these techniques did not result in parallelism *for optimization* being integrated within our modeling framework, they do offer insight regarding what steps may be necessary to achieve this capability. Since Gurobi is not thread safe, there are unique challenges that must be overcome when attempting to parallelize a complex process within Python that iteratively updates model constraints and executes the solver. Additionally, the handling of memory and environment allocation consistently was an obstacle to success for each technique..

Testing of the GurobiPy package with multiple scenarios attempted parallelism from within the commercial solver construct and revealed limitations related to thread control and memory management. These limitations may be lessened via improved memory management such as the use of a built-in sub-process manager to practically distribute memory threads to enable more efficient computing through parallelism. This result motivates further exploration of the GurobiPy package and potential correspondence with Gurobi developers regarding current and future capability. Other future work on parallelism may consider using different solvers and computing packages that offer more environmental and memory control.

Testing the multiprocessing, joblib, and parfor Python libraries also did not improve computational efficiency when routing multiple shipments in parallel. Among the techniques examined, multiprocessing exhibited the greatest control of the environments and assignment of jobs to processors; however, the lack of overhead task management and large amount of parametric data for the underlying problem induced limitations regarding parallelism. Testing revealed the joblib package handled the large data footprint more efficiently but had very little control over both the environment and overhead tasks, ultimately failing to improve the efficiency of solving the lower-level problem. The parfor library was the least capable of the developmental approaches, lacking environmental control and an efficient manner to handle the large data footprint associated with parameterizing the lower-level routing problem.

The recommendations resulting from our study can improve the use of parallelism to solve multiple optimization problems. In a separate research thread, Sauk et al. (2020) recently examined GPU-based optimization algorithms to solve linear programs in parallel. We contend that the future of optimization will entail both types of parallelism – across and within optimization problems – and will allow for better use of high performance computing (HPC) machines that leverage a distributed computing cluster and resource manager, e.g., the Oracle Grid Engine (ORE) or Terascale Open-source Resource and QUEue Manager (TORQUE).

REFERENCES

Gurobi Optimization. "gurobipy 10.0.2: Python interface to Gurobi." PyPI, 13 June 2023, https://pypi.org/project/gurobipy/. Accessed 5 August 2023.

Gurobi Optimization. "Multiple Scenarios", 2023, https://www.gurobi.com/documentation/9.1/refman/multiple_scenarios.html. Accessed 31 August 2023.

Heimes, Christian. "Multiprocessing 2.6.2.1: Backport of the multiprocessing package to Python 2.4 and 2.5." PyPI, 30 July 2009, https://pypi.org/project/multiprocessing/. Accessed 31 August 2023.

Pomp, Wim. "Parfor 2023.8.2: A Package to mimic the Use of parfor as done in Matlab." PyPI, 16 August 2023, https://pypi.org/project/parfor/. Accessed 31 August 2023.

Sauk, Benjamin F. GPU algorithms for large-scale optimization. Dissertation. Carnegie Mellon University, 2020. https://doi.org/10.1184/R1/13194722.v1. Accessed 29 August 2023.

Varoquaux, Gael. "Joblib 1.3.2: Lightweight pipelining with Python functions." PyPI, 9 August 2023, https://pypi.org/project/joblib/. Accessed 31 August 2023.