A Comparison of Algorithms for Solving the Multiagent Simple Temporal Problem

James C. Boerkoel Jr. and Edmund H. Durfee
Computer Science and Engineering
University of Michigan, Ann Arbor, MI 48109, USA
{boerkoel, durfee}@umich.edu

Abstract

The Simple Temporal Problem (STP) is a popular representation for solving centralized scheduling and planning problems. When scheduling agents are associated with different users who need to coordinate some of their activities, however, considerations such as privacy and scalability suggest solving the joint STP in a more distributed manner. Building on recent advances in STP algorithms that exploit loosely-coupled problem structure, this paper develops and evaluates algorithms for solving the multiagent STP. We define a partitioning of the multiagent STP with provable privacy guarantees, and show that our algorithms can exploit this partitioning while still finding the tightest consistent bounds on timepoints that must be coordinated across agents. We also demonstrate empirically that our algorithms can exploit concurrent computation, leading to solution time speed-ups over state-of-the-art centralized approaches, and enabling scalability to problems involving larger numbers of loosely-coupled agents.

Introduction

A person must often develop her schedule with a local, myopic view of how it will interact with the schedules of other individuals. For example, consider a research group consisting of graduate students and a professor. In a given week, each student may need to schedule an individual meeting with the professor, the group as a whole may need to meet once, and the students may need to coordinate over the use of some devices available in the lab. Now suppose each group member enters the week with a tentative, rough schedule of the week: which meetings, social events, personal events, etc. to attend, plus an ordering over when to achieve the events. The problem of determining if these tentative schedules are mutually consistent is an example of a Simple Temporal Problem (STP) (Dechter, Meiri, and Pearl 1991).

Any of several approaches could be taken to determine the consistency of this STP. One is to gather all members' scheduling constraints, and solve the corresponding STP in a centralized fashion. However, this would require that each person reveal his full schedule, which could include doctor's appointments, visits to the parking ticket office, or daily dates with a favorite afternoon soap opera – some information that people may like to keep private. It may also require accumulating a large collection of information that would be daunting to manage. Relative to this illustrative example, the importance of time-critical coordination and privacy can be even more pronounced in military and health care applications, where disclosure of private information or an inability to concurrently calculate and execute a schedule may have significant, adverse effects.

While technology for solving centralized STPs exists, this paper develops a new multiagent approach for computational agents to use in assisting human users both in managing their personal schedules and also in ensuring consistency across the more global STP of which they are a part. Our multiagent approach has several important benefits. First, it respects users' privacy because each scheduling agent reveals only the information necessary to coordinate over a joint constraint with another agent. Thus, agents can identify consistent group meeting times without revealing doctor's appointments or private meetings, even though these other events may indirectly impact when the group can meet. Second, by maintaining this privacy, each agent also retains control over private events, allowing its user to autonomously decide, for example, to skip a television program in favor of work. Finally, a multiagent approach has the advantage of parallel computation, which could speed up the overall solve time. Not only does this avoid the up-front overhead of centralizing a problem that is inherently distributed, but also exploits the near independence of local scheduling problems.

One of the strengths of the STP formulation is that it can be solved to model *sets* of feasible schedules that are more robust to small scheduling disturbances than a single, fully instantiated schedule would be (Cesta and Oddi 1996). This paper describes extending these methods to establish feasible sets of joint schedules across multiagent STPs. Both Smith *et al.* (2007) and Hunsberger (2002) use STPs to increase the resilience of multiagent schedules, not by establishing the full set of joint schedules as we propose, but instead by exploiting sets of individual agents' schedules to absorb minor disturbances within an agent's schedule. In fact, Hunsberger's approach proactively pares down a consistent set of multiagent schedules into decoupled sets of agent schedules by imposing additional local constraints with respect to a common reference point. For

example, if Ann must complete a task before Bill can begin his task, both Ann and Bill can decouple their schedules if Ann agrees to complete her task no later than 3:00 and Bill agrees to begin his task no earlier than 3:00.

This paper's contributions include (1) a novel partitioning of a distributed STP into components that have (2) well-defined, provable properties. Further, this paper contributes (3) novel algorithms for exploiting this partitioning with varying degrees of distributed computation, and finally, (4) an evaluation of these algorithms with respect to non-concurrent computation. This paper proceeds with sections summarizing current approaches for solving STPs, defining the multiagent STP along with proving properties about a partitioning over this STP, describing novel algorithms for establishing multiagent STP consistency, evaluating these algorithms empirically, and summarizing our contributions and a description of our future directions.

Background

The Simple Temporal Problem (STP) consists of a set of timepoint variables, V, and a set of temporal difference constraint edges, E. Each timepoint variable represents an event, and has an implicit, continuous numeric domain. Each temporal difference constraint is of the form $v_i - v_j \in [-B_{ji}, B_{ij}]$, where v_i and v_j are distinct timepoints, and B_{ij} ($\geq v_i - v_j$) and B_{ji} ($\geq v_j - v_i$) are bounds on the difference between v_i and v_j. Every STP has a corresponding *distance graph*, where each timepoint is a vertex (also called node) and there exists an edge for every temporal differences constraint $v_i - v_j \in [-B_{ji}, B_{ij}]$ from v_j to v_i labeled by the bounds interval $[-B_{ji}, B_{ij}]$. Our use of V (vertices) and E (edges) to notate a STP is based on this relationship to distance graphs.

A STP is *consistent* if there exist no negative cycles in the corresponding distance graph. A consistent STP contains at least one *solution*, which is a feasible assignment of specific time values to timepoint variables to form a *schedule*. In many cases, it is desirable to model the entire set of feasible solutions. A *decomposable* STP does this by establishing the tightest bounds on timepoint variables such that (1) no feasible scheduling assignments are eliminated and (2) any assignment of a specific time to a timepoint variable that respects these bounds can be extended to a solution. A decomposable STP instance is extended to a full solution through an iterative cycle of timepoint assignment followed by propagation of this assignment to reestablish decomposability. Using a decomposable STP instance allows a scheduling agent to naturally provide a range of times for each user event such that, if the user executes the event at one of its suggested times, the agent can refine the ranges of remaining events to guarantee a successful schedule.

Figure 1 shows an example distance graph corresponding to the STP formed by two research group members' scheduling problems. This STP includes start and end time events (ST,ET) for a study session (SS), a take-home exam (EXAM), a group project (GP1), and a research paper (RP) for agent 1, and a programming assignment (PA), homework assignment (HW), group project (GP2), and a run

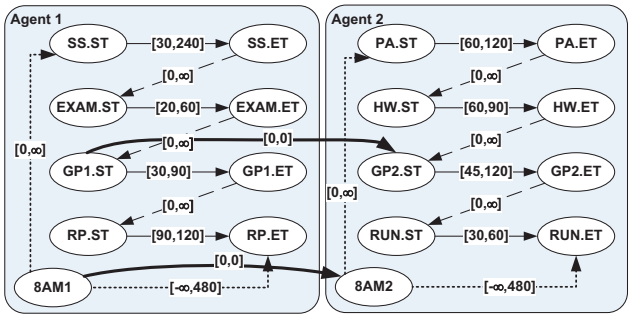

Figure 1: An example STP for two group members

(RUN) for agent 2. In this example, solid edges represent minimum/maximum duration constraints, dashed edges represent precedence constraints, dotted edges represent an overall makespan constraint, and bold edges that span both agents (interagent constraints) represent synchronization constraints. The two agents are synchronized so that their users can exchange project deliverables as they start working on the group project and so that 8AM represents the same reference timepoint (synchronized clocks).

Full-Path Consistency (FPC) determines the consistency of an STP instance in $O(|V|^3)$ time by applying an all-pairs-shortest-path algorithm such as Floyd-Warshall to the distance graph to find the corresponding *d-graph* (Dechter, Meiri, and Pearl 1991). A d-graph is a complete, decomposable distance graph. In this case, decomposability is calculated by tightening the bounds on each edge, e_{ij}, to represent the tightest possible path between v_i and v_j ($B_{ij} \leq B_{ik} + B_{kj} \forall i, j, k$) and verifying that there are no negative paths (if $B_{ij} + B_{ji} \geq 0 \; \forall \; v_i \neq v_j$).

Xu and Choueiry (2003) were the first to recognize that Partial Path Consistency (PPC) (Bliek and Sam-Haroud 1999) is sufficient for establishing decomposability on an STP instance. Applying PPC-based algorithms to calculate STP decomposability requires a triangulated or chordal distance graph representation. A triangulated graph is one whose largest non-bisected cycle is a triangle (of length three). PPC-based algorithms operate by establishing path consistency for each such triangle, thus calculating the tightest possible bounds for each edge in the triangulated graph. Depending on constraint structure, the number of edges in the triangulated graph may be much smaller than the number of edges in the complete d-graph, and so PPC-based algorithms may establish STP decomposability much more quickly than FPC-based algorithms can. If a user is interested in learning the tightest bounds between two timepoint variables that are not mutually constrained (and thus have no edge in the distance graph), the scheduling agent can add this edge explicitly prior to triangulation to ensure its inclusion in the STP decomposability calculation.

Conceptually, a graph is triangulated by the process of considering vertices and their adjacent edges, one-by-one, adding edges between neighbors of the vertex if no edge previously existed, and then eliminating that vertex from further consideration, until all vertices are eliminated. The

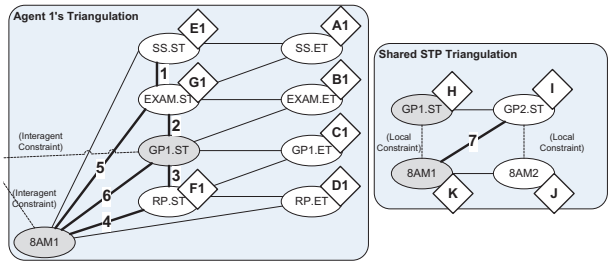

Figure 2: The triangulation process occurring on Agent 1's problem (left) followed by triangulation of the shared STP (right) for the example in Figure 1.

set of edges that are added during this process are called *fill edges* and the order in which timepoints are eliminated from consideration is referred to as an *elimination order*. Elimination orders are often chosen so as to minimize the total number of fill edges. While, generally speaking, finding the minimum triangulation of a graph is NP-complete, heuristics such as the *minimum degree* (vertex with fewest edges) and *minimum fill* (vertex that adds fewest fill edges) are used to approximately minimize triangulations (Kjaerulff 1990).

Figure 2 illustrates triangulation as applied to the example in Figure 1. During this process, edge directionality is ignored. For illustrative purposes, we triangulate agent 1's subproblem first, waiting to eliminate timepoints involved with interagent constraints until the local problems have been completely triangulated. Figure 2 (left) demonstrates the triangulation process on agent 1's STP using the elimination order captured in the labeled diamonds (A1, B1, C1, D1, E1, F1, G1). We label the fill edges (bold) added during the process with the order in which the edges were added. By symmetry, we could triangulate agent 2's local STP in the same manner, leaving the shared STP (with Agent 1's duplicated components shaded) in Figure 2 (right). This particular triangulation results in 6 *new* local fill edges for each agent and one *new* interagent edge for a total of 30 intraagent edges (15 for each agent) and 3 interagent edges (not including the two duplicated edges). Once complete, PPC-based algorithms calculate the tightest path for each of these 33 edges. FPC-based algorithms, on the other hand, calculate the d-graph, thus determining the tightest path for each edge in a fully connected distance graph (in this case, $18 \times 17 \div 2 = 153$ total edges), making FPC much slower than PPC in practice.

Xu and Choueiry's algorithm ΔSTP (2003) processes and updates a queue of all potentially inconsistent triangles (Δ) from the triangulated graph. Alternatively, in their algorithm P3C, Planken, de Weerdt, and van der Krogt (2008) sweep these triangles in a systematic order, resulting in an improved performance (over ΔSTP's $O(|\Delta|^2)$) of $O(|\Delta|)$. While in the worst case, a triangulated STP instance could have as many as $O(|V|^3)$ triangles, in practice, triangulated graphs tend to exploit sparse constraint structures, leading to far fewer triangles than the fully-connected d-graph and much lower expected case complexity than FPC. We point the reader to each of these respective works for more details on centralized PPC algorithms.

Our Multiagent STP Partitioning

Figure 1 provides an example of a multiagent STP. Intuitively, we define a multiagent STP, S, as the union over agents' local STPs. More formally, we define agent i's local STP, S^i, as the tuple $\langle V^i, E^i \rangle$. V^i is partitioned into V^i_A, the set of timepoint variables agent i is responsible for assigning (in Figure 1, V^1_A includes the timepoints in the shaded area labeled Agent 1), and V^i_X, the set of timepoints external to agent i but involved in agent i's interagent constraints ($V^1_X = \{GP2.ST, 8AM2\}$). Note, the sets V^i_A for all agents i are defined to partition the global set of timepoint variables, V. Additionally, E^i is partitioned into the set E^i_N, the (iNternal) *intraagent edges* (E^1_N = the edges included in Agent 1's shaded region) and the set E^i_X, the (eXternal) *interagent edges* ($E^1_X = E^2_X$ and consists of the bold edges spanning the agents).

An important insight that we exploit in our work is that we can further partition the local STPs into shared and private components that have well-defined properties. Figure 2 provides an intuitive glimpse into this, where everything agents 1 and 2 need to know about each other is contained within the Shared STP box. More formally, we partition V^i_A into two sets: V^i_{AP} are agent i's *private timepoints* (which do not appear in V^j_X for any agent j) and V^i_{AS} are agent i's *shared timepoints* (which appear in V^j_X for some other agent(s) j). In our running example, $V^1_{AP} = V^1_A - \{GP1.ST, 8AM1\}$ (all non-shaded timepoints in Figure 2 (left)), and $V^1_{AS} = \{GP1.ST, 8AM1\}$ (Agent 1's shaded timepoints, which also appear in the shared STP). Similarly, we partition E^i_N into two sets: E^i_{NP} is the set of *private edges*, or edges that have at least one endpoint in the set V^i_{AP} (E^1_{NP} corresponds to all edges except edge 6 in Agent 1's triangulation), and E^i_{NS} is the set of *shared intraagent edges* whose endpoints are contained within the set V^i_{AS} (Agent 1's edge 6, which also appears in the Shared STP).

We are now able to define agent i's *private STP*, S^i_P, as the tuple $\langle V^i_A, E^i_{NP} \rangle$, where both V^i_A and E^i_{NP} have already been defined. Additionally, we can define the multiagent *shared STP*, S_S, as the tuple $\langle V_S, E_S \rangle$, where the set of *shared timepoints*, $V_S = \cup_i V^i_{AS}$ (notice $v \in V^j_X$ will be included in V^i_{AS} for some i), and where the set of *shared edges*, $E_S = \{\cup_i E^i_{NS}\} \cup \{\cup_i E^i_X\}$. Figure 2 displays both the shared STP (right), and Agent 1's private STP (left).

Before any coordination occurs in Figure 1, notice that Agent 1 is already aware of Agent 2's timepoints $GP2.ST$ and $8AM2$ due to the shared interagent constraints. Further, if, after eliminating all private timepoints (Figure 2 left), Agent 1 also eliminates $GP1.ST$, it will create and bound an interagent edge between $8AM1$ and $GP2.ST$ (Figure 2 right). Then, if Agent 1 also eliminates $8AM1$, it will have inferred the existence of and bounds on an edge (Agent 2's analogue of Agent 1's edge 6) between $GP2.ST$ and $8AM2$, two timepoints assignable by Agent 2. The question becomes: can Agent 1 continue this process to draw inferences about Agent 2's *private* timepoints and edges?

Obviously, any coordination between agents' activities has some inherent privacy costs. However, we now prove that these costs are limited to the shared timepoints and edges between them. Theorem 1 *guarantees* that Agent 2 will *not* be able to infer the existence of, the number of, or bounds on private activities of Agent 1 (to study, take an exam, etc.) that influence the start time of the group project.

Theorem 1. *No agent can infer the existence of or bounds on another agent's private edges, or subsequently the existence of private timepoints, from the shared STP.*

Proof. First, we prove that the existence and bounds of a private edge cannot be inferred from the shared STP. Assume agent i has a private edge, $e_{xz} \in E^i_{NP}$. By definition, at least one of v_x and v_z is private; WLOG assume $v_x \in V^i_{AP}$. For every pair of edges e_{xy} and e_{yz} that are capable of forming a triangle that implies e_{xz}, regardless of whether v_y is shared or private, $v_x \in V^i_{AP}$ implies $e_{xy} \in E^i_{NP}$ is private. Hence, any pair of edges capable of implying a private edge must also contain at least one private edge. Therefore, a private edge cannot be inferred from shared edges alone.

Now, since an agent cannot extend its view of the shared STP to include another agent's private edge, it cannot infer another agent's private timepoints.

□

Theorem 1 confirms that, while Agent 1 can infer the shared edge between $GP2.ST$ and $8AM2$, it can infer nothing further about Agent 2's timepoints or edges.

Multiagent STP Algorithms

This section presents three algorithms: a centralized, a partially-centralized, and a distributed algorithm, all of which exploit our partitioning to achieve varying levels of computational concurrency.

A Centralized Algorithm

The original P3C takes, as input, a variable elimination ordering and corresponding triangulated STP instance. P3C then sweeps through timepoints in elimination order, propagating the implications of pairs of edges to the edge opposite of the timepoint for each triangle for which it is a part. Once this forward sweep is complete, P3C revisits each triangle in reverse order, tightening all edges to reflect the tightest possible path between each pair of timepoint variables.

Our new triangulating version of P3C, ΔP3C (Algorithm 1), adapts the original P3C algorithm to foster its incorporation into multiagent algorithms. ΔP3C, like P3C, operates in two stages (ΔP3C-1 and ΔP3C-2) and takes as input an STP instance, where we assume that if $e_{ij} \in E$ so is e_{ji}. However, ΔP3C does *not* require the input STP instance to be triangulated or its associated elimination ordering. Instead, it triangulates the STP on the fly. The key insights that allow this modification are (1) that ΔP3C-1 can construct the variable elimination order *during* execution by applying the SELECTNEXT procedure, which heuristically chooses the next timepoint, v_k, to eliminate (line 4) and (2) as ΔP3C-1 considers the implications of each pair of temporal difference constraints involving the removed timepoint variable, it necessarily considers the exact fill edges that the triangulation process would have added. Thus, we introduce procedure JOINNEIGHBORS (line 7), which not only propagates the implications of the eliminated timepoint's constraints forward, but also adds any newly created fill edges (between v_k's non-eliminated neighbors, $N(v_k)$, line 6) to E. In line 8, ΔP3C-1 pushes each triangle it creates onto a stack, so that ΔP3C-2 can retighten each triangle (using the TIGHTENTRIANGLE procedure, line 16) in reverse elimination order.

Algorithm 1 TRIANGULATING-P3C (ΔP3C)

Input: An STP instance $S = \langle V, E \rangle$, and V_E, the elimination subset of V.
Output: The PPC network of S or INCONSISTENT

1: **ΔP3C-1**(S, V_E)**:**
2: $\Delta \leftarrow$ new, empty stack of triangles
3: **while** $V_E \cap V \neq \{\}$ **do**
4: $v_k \leftarrow$SELECTNEXT$(V \cap V_E)$
5: V.remove(v_k)
6: **for all** $v_i, v_j \in N(v_k) i \neq j$ **do**
7: $E \leftarrow E \cup$ JOINNEIGHBORS(v_k, v_i, v_j)
8: **return** INCONSISTENT **if** $(B_{ij} + B_{ji} < 0)$
9: Δ.push(v_i, v_j, v_k)
10: **end for**
11: **end while**
12: $V \leftarrow V \cup V_E$
13: **return** Δ
14: **ΔP3C-2**(Δ) :
15: **while** Δ.size() > 0 **do**
16: $t \leftarrow \Delta$.pop()
17: TIGHTENTRIANGLE(t)
18: **end while**
19: **return** S

JOINNEIGHBORS(v_k, v_i, v_j): Creates edges, e_{ij} and e_{ji}, if they do not already exist, initializing the weights to ∞. Then tightens the bounds of these edges using the rule $B_{ij} \leftarrow \min(B_{ij}, B_{ik} + B_{kj})$. Returns the set of any edges that are created during the process.

TIGHTENTRIANGLE(v_i, v_j, v_k): Tightens any of the triangle edges that need to be tightened using the rule $B_{ij} \leftarrow \min(B_{ij}, B_{ik} + B_{kj})$. Returns the set of any edges that are tightened during the process.

Incorporating the triangulation process into the ΔP3C algorithm reduces the problem of distributing both the P3C and graph triangulation algorithms to that of distributing the execution of the ΔP3C algorithm alone. We also adjust P3C so that we can control the exact subset of timepoint variables, $V_E \subseteq V$, to consider eliminating (lines 3-5). Applying ΔP3C to an STP instance S with $V_E = V$ is semantically identical to applying the original P3C algorithm to S_T and variable elimination ordering o_T, where o_T is the elimination order formed by applying ΔP3C to S, and S_T is the triangulated version of S corresponding to o_T. Our centralized algorithm for solving the multiagent STP, CΔP3C, is now conceptually very simple: aggregate agent subproblems, S^i, into one, centrally-located problem, S, apply ΔP3C with $V_E = V$ to get S', and then redistribute, to each agent, its portion, $S^{i\prime}$, of the tightened edges of the decomposable solution.

A Partially-Centralized Algorithm

By centralizing the problem, not only does an agent reveal its entire private STP, but it now must wait for the central solver. However, in Figure 2, each agent independently triangulated a portion of the STP by eliminating its private timepoints before the agents had to coordinate to triangulate the shared STP. Each agent independently created seven triangles, and coordinated to create two more shared triangles. Our partially-centralized algorithm generalizes this idea so that each agent can *independently* eliminate its private timepoints and tighten its triangles, thus limiting the need for centralization to only the shared STP.

In our partially-centralized algorithm PCΔP3C (Algorithm 2), each agent starts by applying ΔP3C-1 on its STP and set of private timepoints (lines 1-2). Then it sends the shared portion of its STP to a centralized *coordinator* (line 3). The coordinator blocks until it receives the entire shared STP (line 5). The coordinator then applies ΔP3C to the shared STP, which completes the triangulation of the entire multiagent STP (lines 6-7) and fully tightens all shared triangles (line 8). The coordinator then sends each agent its updated portion of the shared STP (line 9). Each agent updates the shared portion of its STP (line 11), and finishes tightening the triangles created during elimination of its private timepoints (line 12) before returning the consistent S^i instance (line 13).

Algorithm 2 PART. CENT. ΔP3C (PCΔP3C)

Input: Agent i's local STP instance $S^i = \left\langle V^i, E^i \right\rangle$, and the id of the coordinator $coordID$
Output: The PPC network of S^i or INCONSISTENT
1: $\Delta^i \leftarrow \Delta$P3C-1$(S^i, V^i_{AP})$
2: **return** INCONSISTENT if ΔP3C-1 does
3: SEND$(coordID, \left\langle V^i_{AS}, E^i_{NS} \cup E^i_X \right\rangle)$
4: **if** $(i = coordID)$ **then**
5: $S_S \leftarrow \cup_i$ BLOCKRECEIVE(Agent j, S^j_S) $\forall j$
6: $\Delta_S \leftarrow \Delta$P3C-1$(S^S, V^S)$
7: **return** INCONSISTENT if ΔP3C-1 does
8: ΔP3C-2(Δ_S)
9: SEND(Agent j, S^j_S) $\forall j$
10: **end if**
11: $S^i \leftarrow$ BLOCKRECEIVE$(coordId, S^i_S)$
12: ΔP3C-2(Δ^i)
13: **return** S^i

Proposition 1. *PCΔP3C correctly applies ΔP3C to the multiagent STP.*

Proof (Sketch). We first prove that each agent i correctly applies ΔP3C-1 to V^i_{AP}. The idea is that in order for each agent i to correctly eliminate timepoint $v_x \in V^i_{AP}$, it must know current, correct bounds for any edges involving v_x. The only way for an edge involving v_x to be added or updated is if a neighboring timepoint of v_x is eliminated. However, since by definition, the neighbors of v_x are a subset of V^i_{AP} and since agent i never eliminates a timepoint in V^i_{AS} before v_x (which could introduce a new interagent edge), agent i can consistently eliminate v_x without any impact from (and without impacting) any other agent j.

We then prove that the coordinator correctly applies ΔP3C to the shared STP. The idea is that, while eliminating timepoints in V^i_{AP} may affect shared edges, by waiting for all such updates from each agent, the coordinator will be sufficiently aware of all edges that have been added or updated during agents' local triangulation. Once this handoff completes successfully, the shared STP is simply a new STP instance on which the coordinator applies ΔP3C.

Finally, we prove that each agent i correctly applies ΔP3C-2 to Δ^i. The idea here is that some of the triangles that agent i creates applying ΔP3C-1 to V^i_{AP} share edges with triangles in the shared STP. However, given that the coordinator correctly achieves decomposability on S_S, each of these shared edges is guaranteed to be the tightest possible consistent with the overall multiagent STP S. Hence, similarly to the application of ΔP3C-1 which created this stack, an agent i can tighten its triangle stack Δ^i independently of any other agent j tightening its triangle stack Δ^j

□

A Fully Distributed Algorithm

In our partially-centralized algorithm, the coordinator waits for each agent to triangulate its private portion of the multiagent STP, and in turn, each agent must wait for the coordinator to send back its tightened edges. In the example in Figure 2, upon triangulating its private timepoints, agent 1 could optimistically proceed with eliminating, for example, $GP1.ST$. However, unlike when Agent 1 eliminated its private timepoints, it now must worry about Agent 2 performing eliminations that affect how Agent 1 should triangulate the shared STP. For example, suppose, unbeknownst to Agent 1, Agent 2 has already eliminating $GP2.ST$. In this case, Agent 1 will be eliminating $GP1.ST$ assuming $N(GP1.ST) = \{8AM1, GP2.ST\}$, when in reality $N(GP1.ST) = \{8AM1, 8AM2\}$. Thus, Agent 1's optimistic elimination of $GP1.ST$ will be inconsistent.

Meanwhile, suppose that Agent 2 has eliminated all its timepoints, and now would like to proceed with its backward sweep to tighten all its triangles. Instead of waiting for Agent 1 to finish its forward sweep, it can proceed, optimistically trusting its calculation of bounds over shared edges (in this case [120,405] for both $GP1.ST-8AM1$ and $GP2.ST-8AM2$). So when Agent 1 subsequently tightens these bounds to [120,360], Agent 2 will have to retighten any of its triangles that depend on the new, tighter upper bound of 360. Agent 2's optimism pays off for all triangles dependent on only the lower bound, since it already tightened triangles using the correct lower bound. Note, the alternative to this optimistic computation is idleness, so optimism has no time costs, even when triangles must be retightened.

In our distributed algorithm DΔP3C (Algorithm 3), each agent starts by applying ΔP3C-1 on its STP and set of private timepoints (lines 1-2). An agent then continues to optimistically eliminate its shared timepoints in lines 4-22, using a copy of its edge set E and triangle stack Δ to recover in case its optimism is misguided. The agent, after selecting a variable, v_k, to eliminate (tentatively) and computing what edges to add and/or update, then calls REQUESTELIMINATIONORDERLOCK(line 12) to obtain write permission on an

object containing the shared timepoint elimination order and added/updated edges. First, the agent must confirm that no neighbors of v_k have been eliminated in the meantime (line 13). If not, its bounds on all edges involving v_k are current, and so it can commit to eliminating the timepoint (lines 14-16). Otherwise, it extracts all updates to any affected edges (UPDATEEDGES) to revise its local STP and abandons the changes calculated in $\tilde{E}^i$ and $\tilde{\Delta}^i$. Whether it succeeds at eliminating a timepoint or not, the agent loops until all its remaining shared timepoints have been eliminated.

We purposely label the backward sweep of DΔP3C as ΔP3C-MAINTENANCE, because this procedure could naturally be extended to a multiagent STP consistency maintenance algorithm that is capable of processing dynamic tightening of bounds as users execute their schedules. Here, an agent tightens triangles in the order in which they appear in its stack (lines 29-30). An agent can receive notice of a new, tighter bound on an edge from RECEIVEEDGEUPDATE(). If an agent receives any such updates, it inserts all adjacent triangles into their proper locations in the stack, if they are not already there, by calling INSERTADJACENTTRIANGLES() on the set of updates (line 27). We define a triangle to be *adjacent* to an edge e_{ij} if a triangle was created upon elimination of some timepoint, v_k, such that $v_i, v_j \in N(v_k)$ at the time of elimination. The agent also records the set of edges that it tightens (line 30), so that it can update other agents by calling BROADCASTANYSHAREDEDGEUPDATES(), which filters out and broadcasts shared updated edges, and also ensures that subsequently-affected local triangles are inserted into the triangle stack.

Proposition 2. *DΔP3C correctly applies ΔP3C to the multiagent STP.*

Proof (Sketch). We borrow much of the intuition for the triangulation portion (ΔP3C-1) of the algorithm from the PCΔP3C proposition. However, an agent eliminating a shared timepoint is no longer independent from the actions of other agents. So the idea here is that an agent assumes it has sufficient knowledge to eliminate a shared timepoint, but only commits to it once it receives a lock on the elimination ordering – that is, once it confirms that no other agents' eliminations will affect this timepoint elimination.

The basic idea for the triangle tightening portion of the algorithm is similar to that of the ΔSTP algorithm. That is, every triangle is processed at least once, and any triangle that becomes 'untightened' (due to a tightened edge bound), will be inserted into the stack to be tightened again. In addition to this intuition, in the worst case (where every tightened bound requires retightening every adjacent triangle), the triangles of the multiagent STP will be processed in the same order as they would be in the PCΔP3C algorithm. Even if DΔP3C cannot avoid this worst-case computation performance, it will calculate the same decomposable STP instance as PCΔP3C, and hence, ΔP3C.

□

Algorithm 3 DISTRIBUTED ΔP3C (DΔP3C)

Input: Agent i's local STP instance $S^i = \left\langle V^i, E^i \right\rangle$
Output: The PPC network of S^i or INCONSISTENT

1: $\Delta^i \leftarrow \Delta$P3C-1$(S^i, V^i_{AP})$
2: **return** INCONSISTENT if ΔP3C-1 does
3: $V^i_E \leftarrow V^i_{AS}$.copy()
4: **while** $V^i_E \cap V^i \neq \{\}$ **do**
5: $\quad \tilde{E}^i \leftarrow E^i$.copy(), $\tilde{\Delta}^i \leftarrow \Delta^i$.copy()
6: $\quad v_k \leftarrow$SELECTNEXT$(V^i \cap V^i_E)$
7: $\quad$ **for all** $v_i, v_j \in N(v_k)$ **do**
8: $\quad\quad \tilde{E}^i \leftarrow \tilde{E}^i \cup$ JOINNEIGHBORS(v_k, v_i, v_j)
9: $\quad\quad$ **return** INCONSISTENT **if** $(B_{ij} + B_{ji} < 0)$
10: $\quad\quad \tilde{\Delta}^i$.push$(v_i, v_j, v_k)$
11: $\quad$ **end for**
12: $\quad o \leftarrow$REQUESTELIMINATIONORDERLOCK()
13: $\quad$ **if** $(o \cap N(v_k) = \emptyset)$ **then**
14: $\quad\quad o$.append(v_k)
15: $\quad\quad V$.remove(v_k)
16: $\quad\quad E^i \leftarrow \tilde{E}^i, \Delta^i \leftarrow \tilde{\Delta}^i$
17: $\quad$ **else**
18: $\quad\quad S^i \leftarrow$ UPDATEDEDGES$(o \cap N(v_k))$
19: $\quad$ **end if**
20: $\quad$RELEASEELIMINATIONORDERLOCK(o)
21: **end while**
22: $V^i = V^i \cup V^i_E$
23: **ΔP3C-MAINTENANCE(Δ^i):**
24: $U \leftarrow$ new updated edge stack
25: **while** Δ^i.size() > 0 **or** PENDINGEDGEUPDATES **do**
26: $\quad U.push($RECEIVEEDGEUPDATES()$)$
27: $\quad \Delta^i$.INSERTADJACENTTRIANGLES(U)
28: $\quad U.clear()$
29: $\quad t \leftarrow \Delta^i$.pop()
30: $\quad U.push($TIGHTENTRIANGLE$(t))$
31: $\quad$BROADCASTANYSHAREDEDGEUPDATES(U)
32: **end while**
33: **return** S^i

INSERTADJACENTTRIANGLES(U): Updates the triangle stack to include any (externally or internally) updated triangle adjacent to updated edges (except the triangle that caused the update) to its specified location in the triangle stack Δ^i.

Evaluation

In this section, we empirically evaluate our hypothesis that the restrictions our algorithms place on agent's timepoint elimination heuristics hurts its performances (in terms of total number of fill edges added). We also evaluate our hypothesis that algorithms exploiting our agent partitioning can lead to reduced non-concurrent computation.

Experimental Setup

We evaluate our algorithms for solving multiagent STPs on randomly-generated STP instances. While real problem instances would allow us to better characterize the performance of our algorithms on naturally structured problems, random problem generation allows us to control the complexity of and the relative private-to-shared timepoint ratio in the composition of problem instances. The random problem generator is parameterized by the tuple $\langle A, T, P, C_{Intra}, C_{Inter} \rangle$ where A is the number of agents, T is the number of timepoint variables per agent, P is the percentage of its timepoints that an agent keeps private, C_{Intra} is the number of local constraints per agent, and C_{Inter} is the total number of interagent constraints. Our default parameter settings are $A = 25$, $T = 25$, $P = 67\%$,

$C_{Intra} = 200$, and $C_{Inter} = 3350$. Using our default parameter settings as a basis, we normalize our results as we vary P by scaling the number of constraints (C_{Intra} and C_{Inter}) so that, in expectation, the complexity of the centralized algorithm is constant (falls within 5% of the complexity of our default settings).

To confirm the significance of our results, we run our experiments using 25 trials (each with a distinct random seed). Our algorithms were programmed in Java, on a 2 GHz processor with 2 GB of RAM. For the purposes of modeling a concurrent, multiagent system, we interrupted each agent after it was given the opportunity to perform one constraint check and send one message, systematically sharing the processor between all agents involved. All our approaches use the minimum fill heuristic (Kjaerulff 1990). Our approaches were applied to connected networks of agents, although intuitively, the performance of any of our algorithms would be enhanced by applying them to disparate agent networks, independently. Finally, all problem instances were generated to lead to consistent, decomposable STP instances to evaluate a full application of each algorithm. In general, however, unlike previous approaches, our algorithms do not require input STPs to be consistent (Hunsberger 2002) or triangulated (Planken, de Weerdt, and van der Krogt 2008).

Impact on Fill Edge Heuristics

The minimum fill heuristic selects the timepoint, from a set of timepoints, that it expects will lead to the fewest fill edges. Since the centralized algorithm, CΔP3C, places no restrictions on this heuristic, we expect it to perform well. PCΔP3C and DΔP3C, on the other hand, both exploit the partitioning by restricting private timepoints to be eliminated prior to shared timepoints. And whereas the coordinator in the PCΔP3C can apply fill heuristics to the set of all shared timepoints, each agent in DΔP3C is restricted to applying this heuristic to only its timepoints. Intuitively, we expect each of these additional restrictions to hurt heuristic performance, that is, to add more fill edges. We test this hypothesis on problems by increasing the proportion of private timepoints (P); the results are displayed in Figure 3.

Overall, the number of fill edges decreases as P increases, since, as constraints become more dense in the private STPs, more triangles are formed inherently, resulting in fewer added fill edges. While, as expected, DΔP3C adds more fill edges than the other two algorithms, surprisingly, the expected number of fill edges (Figure 3, top) added using $C\Delta$P3C and PCΔP3C is nearly indistinguishable. As P nears 1.0, the fill edge curve of DΔP3C eventually approaches that of $C\Delta$P3C, since the restrictions on the heuristic have diminishing impact as interagent constraints grow more sparse.

The specific differences between the expected number of fill edges for CΔP3C and PCΔP3C are statistically insignificant. By performing a paired Student's T-test, however, we find that the number of fill edges is statistically unlikely to come from the same populations. This means that differences do in fact exist. In Figure 3 (bottom), we plot the *ratio* of the number of fill edges for both PCΔP3C and DΔP3C to the number of fill edges generated by CΔP3C.

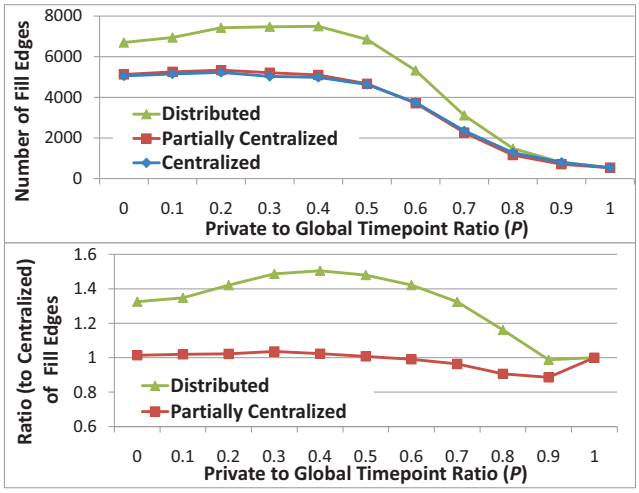

Figure 3: Fill edges *vs.* P (top). Ratio (to centralized) of fill edges *vs.* P (bottom).

This shows that the restrictions imposed by our partitioning of the STP *hurt* when P is low (when most triangles end up being shared), increasing the ratio fill edges up to 5% and *help* when P is high (when most triangles end up being private), decreasing the ratio of fill edges up to 10%. The additional restrictions placed on DΔP3C lead to up to a 50% increase in fill edges and never significantly fewer edges than CΔP3C.

These results are important, since they imply that, before computational concurrency is taken into account, the structural knowledge of our STP partitioning, as shown by the PCΔP3C curve, can reduce the total amount of computation. Clearly, if agents use a method for determining the *best* elimination ordering (an NP-complete problem), the centralized approach would be guaranteed to find it, though these results suggest that centralized heuristics could benefit from using the structural knowledge embedded in our partitioning. While these problems were randomly generated (and it would be easy to generate pathological cases where a centralized view of the problem is critical to heuristic performance), the real-world problems that we are interested in solving with our multiagent scheduling system tend to have more locally-dense, well-structured constraints than those of our randomly-generated problem instances.

Impact on Concurrent Execution

One of the main benefits that we associate with performing a greater amount of computation in a more distributed fashion is that it promotes greater concurrency. The more agents that can be computing at the same time, theoretically, the less time it takes to complete the same amount of computation. In this section, we explore how well our multiagent algorithms can exploit concurrent computation, reporting the number of non-concurrent computational units. Non-concurrent computational units measure the number of cycles it takes to establish global STP PPC, where each agent

is given an opportunity to perform a single constraint check each cycle of computation (although agents may spend this cycle idly blocking on updates from other agents). Since DΔP3C requires a significant number of messages, we separately count the number of computation cycles where at least one agent sends a message.

Figure 4 (top) shows the non-concurrent computation curves for CΔP3C, PCΔP3C, and DΔP3C algorithms, as well as an additional curve that represents the sum of DΔP3C's message and computation cycles. We see that when P is low, PCΔP3C behaves much like CΔP3C, and when P is high, it performs more similarly to DΔP3C. When P is low, DΔP3C, in expectation, performs roughly four times fewer non-concurrent computational units (three when incorporating messages) than CΔP3C and exceeds 22 times speedup (given 25 agents, perfect speedup is 25) for high P values. For both PCΔP3C and DΔP3C, the lack of concurrency is mainly due to the synchrony required to solve the shared STP. As the size of the shared STP shrinks relative to the size of the local STPs, this source of non-concurrency is reduced, resulting in improved performance. In both cases, imbalance in the complexity of individual agent problems prevents the algorithms from achieving perfect speedup.

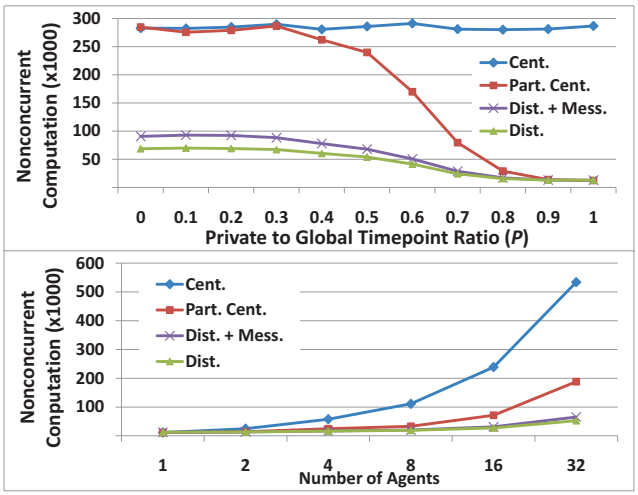

Figure 4: Non-concurrent computation *vs.* P (top). Non-concurrent computation *vs.* A (bottom).

Finally, Figure 4 (bottom) shows the non-concurrent computation as the number of agents grows. We see that the number of non-concurrent constraint checks tends to grow linearly with the number of agents for both CΔP3C and PCΔP3C. For this set of experiments, P was set at 67%, thus PCΔP3C grows about a third as quickly as CΔP3C and has a speedup that hovers around 3. Figure 4 (bottom) also shows that DΔP3C complexity increases more slowerly than the PCΔP3C, and DΔP3C's speedup increases with the number of agents as seen by the widening relative gap between the CΔP3C and DΔP3C curves.

Conclusions and Future Work

In this paper, we defined a partitioning over a multiagent STP that divides agent problems into shared and private components with provable privacy guarantees. We developed three algorithms that trade centralized control for increased privacy and concurrency. We evaluated our algorithms and found the benefits of concurrent computation far exceeded any costs associated with extra fill edges. In fact, the restrictions on elimination ordering that our STP partitioning imposed served to improve heuristic performance when there was sufficient local STP structure. Finally, we empirically demonstrated, on randomly-generated problem instances, that DΔP3C dominates other approaches in terms of non-concurrent computation, by using idle computational cycles to proceed optimistically.

In the future, we wish to extend these results to investigate how agents can react to dynamics within multiagent STPs. Such dynamics could include the addition or removal of constraints or agents, and could take advantage of approaches that efficiently propagate incremental updates to consistent STPs (Cesta and Oddi 1996; Planken 2008). Additionally, we wish to extend our evaluation to real multiagent STP instances. Finally, since STPs are often solved in subroutines for richer planning or scheduling domains, we would like to explore how our alternative algorithms fare embedded in multiagent algorithms for more complex temporal problems.

Acknowledgments

We thank the anonymous reviewers for their comments and suggestions. This work was supported, in part, by the NSF under grant IIS-0534280 and by the AFOSR under Contract No. FA9550-07-1-0262.

References

Bliek, C., and Sam-Haroud, D. 1999. Path consistency on triangulated constraint graphs. In *IJCAI 1999*, 456–461.

Cesta, A., and Oddi, A. 1996. Gaining efficiency and flexibility in the simple temporal problem. In *Int. Workshop on Temporal Representation and Reasoning 1999*.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal constraint networks. *Artificial Intelligence* 49(1-3):61–95.

Hunsberger, L. 2002. Algorithms for a temporal decoupling problem in multi-agent planning. In *AAAI 2002*, 468–475.

Kjaerulff, U. 1990. Triangulation of graphs - algorithms giving small total state space. Technical report.

Planken, L.; de Weerdt, M.; and van der Krogt, R. 2008. P3c: A new algorithm for the simple temporal problem. In *ICAPS 2008*, 256–263.

Planken, L. 2008. Incrementally solving the stp by enforcing partial path consistency. In *PlanSIG 2008*, 87–94.

Smith, S. F.; Gallagher, A.; Zimmerman, T.; Barbulescu, L.; and Rubinstein, Z. 2007. Distributed management of flexible times schedules. In *AAMAS 2007*, 472–479.

Xu, L., and Choueiry, B. Y. 2003. A new efficient algorithm for solving the simple temporal problem. In *4th Int. Conf. on Temporal Logic 2003*, 210–220.