Generating Parallel Execution Plans with a Partial Order Planner

Craig A. Knoblock
USC/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292-6695

May 1994
ISI/RR-94-380

Many real-world planning problems require generating plans that maximize the parallelism inherent in a problem. There are a number of partial-order planners that generate such plans; however, in most of these planners it is unclear under what conditions the resulting plans will be correct and whether the planner can even find a plan if one exists. This paper identifies the underlying assumptions about when a partial plan can be executed in parallel, defines the classes of parallel plans that can be generated by different partial-order planners, and describes the changes required to turn UCPOP into a parallel execution planner. In addition, we describe how this planner can be applied to the problem of query access planning, where parallel execution produces substantial reductions in overall execution.
GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to stay within the lines to meet optical scanning requirements.

Block 1. Agency Use Only (Leave blank).

Block 2. Report Date. Full publication date including day, month, and year. If available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. Denotes public availability or limitations. Include day, month, and year if available (e.g. availability to the public. Enter additional limitations or special markings in all caps (e.g. NOFORN, REL. ITAR).

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all caps (e.g. NOFORN, REL. ITAR).

DOD - See DoD 5230.24, "Distribution Statements on Technical Documents."

DOE - See authorities.


NTIS - Leave blank.

Block 12b. Distribution Code.

DOD - Leave blank.

DOE - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.

NASA - Leave blank.

NTIS - Leave blank.

Block 5. Funding Numbers. To include contract and grant numbers. May include program element numbers(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C - Contract
G - Grant
PE - Program
Project
Element
TA - Task
WU - Work Unit
Accesion No.

Block 6. Author(s). Name(s) of persons responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. (If known)

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with... Trans. of... To be published in... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 13. Abstract. Include a brief (Maximum 200 words) factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code (NTIS only).


Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.
Generating Parallel Execution Plans with a Partial-Order Planner

Craig A. Knoblock
Information Sciences Institute and Department of Computer Science
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292
knoblock@isi.edu

Abstract

Many real-world planning problems require generating plans that maximize the parallelism inherent in a problem. There are a number of partial-order planners that generate such plans; however, in most of these planners it is unclear under what conditions the resulting plans will be correct and whether the planner can even find a plan if one exists. This paper identifies the underlying assumptions about when a partial plan can be executed in parallel, defines the classes of parallel plans that can be generated by different partial-order planners, and describes the changes required to turn UCPOP into a parallel execution planner. In addition, we describe how this planner can be applied to the problem of query access planning where parallel execution produces substantial reductions in overall execution time.

Introduction

There are a wide variety of problems that require generating parallel execution plans. Partial-order planners have been widely viewed as an effective approach to generating such plans. However, strictly speaking, a partially-ordered plan represents a set of possible totally-ordered plans. Just because two actions are unordered relative to one another does not imply that they can be executed in parallel. The semantics of a partially-ordered plan provide that the two actions can be executed in either order. Simultaneous execution requires that potential resource conflicts between unordered actions be made explicit and avoided.

There are numerous partial-order planners presented in the literature, including SIVE (Wilkins 1984), NONLIP (Tate 1976), SNIP (McAllester & Rosenblit 1991), UCPOP (Penberthy & Weld 1992), TWEAK (Chapman 1987), O-PLAN (Currie & Tate 1991), etc. Many of these planners have been used to produce parallel plans but previously no one has precisely described the class of parallel plans they produce, identified their limitations, or described the assumptions made for the parallel plans that they do produce.

This paper focuses on the use of partial-order planning to generate parallel execution plans. First we identify the conditions under which two unordered actions can be executed in parallel. The component missing from many planners is an explicit representation of resources. Second, assuming that the resource constraints have been made explicit we identify the classes of partial execution plans that can be generated using different partial-order planners. Third, we present an implementation of a parallel execution planner based on UCPOP. Fourth, we describe how this planner can be used to generate parallel query access plans. Fifth, we compare the use of a partial-order planner to other approaches to building parallel execution plans. Finally, we review the contributions of the paper and describe some directions for future research.

Executing Actions in Parallel

Classical planners assume that the execution of an action is indivisible and uninterruptible (Weld 1994). This is referred to as the atomic action assumption and stems from the fact that the STRIPS-style representation only models the preconditions and effects of an action. This assumption would appear to make simultaneous execution impossible, since it is unclear from the action model whether any two actions can be executed simultaneously without interacting with one another. This section identifies the conditions under which it is possible to execute two actions in parallel.

The work on parallelizing execution of machine instructions (Tjaden & Flynn 1976) provides some insight on the types of dependencies that arise between actions. Tjaden and Flynn identify three types of dependencies that must be considered in parallelizing machine instructions: procedural, operational, and data. A procedural dependency occurs when one instruction explicitly addresses another instruction and therefore imposes an ordering between the instructions. An operational dependency occurs when there is a resource...
Generating Parallel Execution Plans with a Partial-Order Planner*

Craig A. Knoblock
Information Sciences Institute and
Department of Computer Science
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292
knoblock@isi.edu

Abstract

Many real-world planning problems require generating plans that maximize the parallelism inherent in a problem. There are a number of partial-order planners that generate such plans; however, in most of these planners it is unclear under what conditions the resulting plans will be correct and whether the planner can even find a plan if one exists. This paper identifies the underlying assumptions about when a partial plan can be executed in parallel, defines the classes of parallel plans that can be generated by different partial-order planners, and describes the changes required to turn UCPOP into a parallel execution planner. In addition, we describe how this planner can be applied to the problem of query access planning, where parallel execution produces substantial reductions in overall execution time.

Introduction

There are a wide variety of problems that require generating parallel execution plans. Partial-order planners have been widely viewed as an effective approach to generating such plans. However, strictly speaking, a partially-ordered plan represents a set of possible totally-ordered plans. Just because two actions are unordered relative to one another does not imply that they can be executed in parallel. The semantics of a partially-ordered plan provide that the two actions can be executed in either order. Simultaneous execution requires that potential resource conflicts between unordered actions be made explicit and avoided.

There are numerous partial-order planners presented in the literature, including SIP (Wilkins 1984), NON-LIN (Tate 1976), SNLP (McAllester & Rosenblitt 1991), UCPOP (Penberthy & Weld 1992), TWEAK (Chapman 1987), O-PLAN (Currie & Tate 1991), etc. Many of these planners have been used to produce parallel plans, but previously no one has precisely described the class of parallel plans they produce, identified their limitations, or described the assumptions made for the parallel plans that they do produce.

This paper focuses on the use of partial-order planning to generate parallel execution plans. First, we identify the conditions under which two unordered actions can be executed in parallel. The component missing from many planners is an explicit representation of resources. Second, assuming that the resource constraints have been made explicit, we identify the classes of parallel execution plans that can be generated using different partial-order planners. Third, we present an implementation of a parallel execution planner based on UCPOP. Fourth, we describe how this planner can be used to generate parallel query access plans. Finally, we review the contributions of the paper and describe some directions for future research.

Executing Actions in Parallel

Classical planners assume that the execution of an action is indivisible and uninterruptible (Weld 1994). This is referred to as the atomic action assumption and stems from the fact that the STRIPS-style representation only models the preconditions and effects of an action. This assumption would appear to make simultaneous execution impossible, since it is unclear from the action model whether any two actions can be executed simultaneously without interacting with one another. This section identifies the conditions under which it is possible to execute two actions in parallel.

The work on parallelizing execution of machine instructions (Tjaden & Flynn 1970) provides some insight on the types of dependencies that arise between actions. Tjaden and Flynn identify three types of dependencies that must be considered in parallelizing machine instructions: procedural, operational, and data. A procedural dependency occurs when one instruction explicitly addresses another instruction and therefore imposes an ordering between the instructions. An operational dependency occurs when there is a resource
Parallel Execution Plans

This section identifies the classes of parallel execution plans that can be generated by different planners, assuming that a domain is correctly axiomatized and explicitly represents the resource requirements of the operators. The different types of parallel execution plans can be broken down into several classes, ranging from plans with completely independent actions to those where the actions must be executed in parallel or must overlap in a particular manner in order for the plan to succeed. As the interactions in the plan increase in complexity, the corresponding planners require more sophisticated representations and reasoning in order to generate such plans. In this section we present four classes of parallel execution plans and identify the corresponding planners that can generate that class of plans. These classes are described in order from the most restrictive to the least restrictive.

Independent Actions

The most restricted type of parallel execution plans are those where all of the parallel actions are completely independent of one another.

Two actions are defined to be independent if and only if the effects of the two actions executed simultaneously are the union of the individual effects of the actions done in isolation.

Allen (1991) notes that various partial-order planners, such as NONLIN (Tate 1976), DEVISER (Vere 1983), and SIPE (Wilkins 1984), all “allow simultaneous action when the two actions are completely independent of each other.” While this statement is correct, it is a bit misleading since these planners can generate plans for a less restrictive class of parallel plans. As noted by Horz (1993), since some of the effects of an operator may be unnecessary with respect to the goal and preconditions of other operators, the fact that two operators are unordered in a plan generated by a partial-order planner does not imply that they are independent. A planner that can only generate plans with independent actions is UA (Minton, Bresina, & Drummond 1991), which imposes ordering constraints between any pair of unordered actions that could possibly interact.

Figure 1 illustrates a simple plan with two independent actions. The goal of the plan is to have the table painted red and the chair painted blue. Since the actions of painting the table and painting the chair are independent, they can be executed in parallel.

Independent Actions Relative to a Goal

In a variety of partial-order planners, such as SIPE (Wilkins 1984), SNLP (McAllester & Rosenblitt 1991), and UCP (Penberthy & Weld 1992), the planners enforce the property that two actions can only remain unordered if there is no threat between them. A threat occurs when an operator could potentially delete a con-
Two or UCPOP, defined a threat to include an operator that adds or deletes a relevant condition. This stronger definition of a threat is used to constrain the search space and would prevent some possible parallel plans from being generated.

Independent Subplans Relative to a Goal

Not all partial-order planners enforce the property that two actions can remain unordered only if there are no threats between them. In particular, those planners that implement some form of Chapman's white knight (Chapman 1987) require only that there exist some operator that establishes a given precondition, but do not commit to which operator. More specifically, the white knight operation allows plans with the following conditions: There exists some operator op1 that achieves a goal or precondition $g$. There exists a second operator op2 that possibly deletes $g$. And there exists a third operator op3 that follows op2 and achieves $g$. If we are interested in producing totally-ordered plans, then the white knight operator is not required for completeness. However, the use of the white knight operator allows a planner to generate a slightly more general class of parallel plans.

The planners in this class include TWEAK (Chapman 1987), NONLIN (Tate 1976), OP-PLAN (Currie & Tate 1991), MPI, and MP (Kambhampati 1994). The class of parallel plans produced by these planners are those with independent subplans relative to a goal.

Two subplans are independent relative to a goal $G$ if and only if, for all conditions that are relevant to achieving $G$, the result of executing the subplans in either order is identical to the result of executing the actions simultaneously.

The class of parallel plans that can be generated by the planners in this class, but cannot be generated by the planners in the previous class are those where there are actions that are not independent, but the subplans in which the actions occur are independent.

Figure 3 shows an example plan with independent subplans relative to the goal (adapted from an example in Kambhampati 1994)). In this example, before the table and chair can be painted red, they must be primed, and priming them has a side-effect of painting the floor white. The final goal of the problem is to get the table, chair, and floor all painted red. Notice that the action of painting the chair interacts with painting the table, since they both change the color of the floor. Similarly, painting the table interacts with painting the chair. Despite these potential interactions, the floor will still be painted red at the end of the plan since the table and chair must be painted after they are primed. Solving this problem requires the use of the white knight operator to produce the parallel plan since the plan does not state which painting operation will be used to achieve the final goal of making the floor red.

The implementation of the white knight, which allows a planner to generate this more general class of parallel plans, also makes it difficult to extend the operator language to efficiently handle more expressive constructs, such as conditional effects and universal quantification (Chapman 1987). These more expressive language constructs are often required for representing and solving real problems.
Interacting Actions

The most general class of parallel plans are those where the parallel actions interact in some way. Two actions may need to be executed in parallel or two actions may need to overlap in a particular manner in order for the plan to succeed. For example, if the final goal was to get the chair blue, the table yellow, and the floor green and there was no green paint, we could paint the table and chair simultaneously. To handle these cases requires the introduction of an explicit representation of time, such as that provided in temporal planning systems (Allen et al. 1991; Penberthy 1993). However, in this paper we are interested in the more restricted case where we would like to execute actions in parallel to take advantage of the possible parallelism to reduce the total execution time, not because the solution requires parallelism to solve the problem.

Parallel Execution Planning in UCPOP

We used the ucpop planner (Penberthy & Weld 1992; Barrett et al. 1993) to build a parallel execution planner. The analysis in the previous section showed that ucpop can produce the class of plans with actions that are independent relative to a goal. For the specific application described in the next section, this restriction does not prevent the system from finding any solutions. The changes to ucpop that were required were to add explicit resource definitions to the operators, to modify the planner to enforce the resource constraints, and to construct an evaluation function to estimate the cost of the parallel plans.

The resource requirements of the operators are made explicit by augmenting each operator with a resource declaration. An example operator with a resource declaration is shown in Figure 4. This operator describes the action of moving data from one data source to another and declares the data source from which the data is being moved as a resource. The purpose of this declaration is to prevent one operator from being executed in parallel with another operator that requires the same database.

In order to avoid resource conflicts, we modified the planner to ensure that if two operators require the same resource, then they are not left unordered relative to one another. In SIPE this is done with a critic that checks for resource conflicts and then imposes ordering constraints when conflicts are found. In UCPOP, we added a check to the planner such that every time a new action is added to the plan, the planner checks for potential resource conflicts with any other operator that could be executed in parallel. Any conflicts discovered are added to the list of threats that must be removed before the plan is considered complete. Using the search control facility in ucPOP, these conflicts can be resolved immediately or delayed until later in the planning process.

Since efficiency is the primary motivation for generating parallel plans, we constructed an evaluation function that can be used to find plans with low overall execution time. Since this evaluation function underestimates the cost of the parallel plan, the planner can use a best-first search to find the optimal plan. This evaluation function takes into account that the cost of executing two actions in parallel will be the maximum and not the sum of the costs. The space of parallel execution plans may be quite large, so domain-specific control knowledge may be necessary to search this space efficiently.

The evaluation function to determine the execution time of a parallel execution plan is implemented using a depth-first search. The search starts at the goal node and recursively assigns a cost to each node in the plan. This cost represents the total cost of execution up to and including the action at the given node. The cost is calculated by adding the cost of the action at the node to the maximum cost of all the immediately prior nodes. Once the cost of the plan up to a node has been computed, we store this value so it will only need to be calculated once. Since each node (n) and each edge (e) in the graph is visited only once, the complexity of evaluating the plan cost is O(max(n,e)).
Parallel Query Access Plans

There are two general characteristics of a domain where the use of a partial-order parallel-execution planner will be useful and effective. First, it is applicable to those domains where the actions could be executed serially, but the overall execution time can be reduced by executing some of the actions in parallel. Second, it will be most useful in those domains where the choice of the operations determines or limits the overall execution time of the plan. As such, the plan generation and scheduling cannot be done independently since this would potentially result in highly suboptimal plans.

We applied the parallel execution planner to a query access planning problem that involves multiple distributed information sources (Knoblock & Arens 1994). In this domain, a plan is produced that specifies how to retrieve the requested set of data. This requires selecting sources for the data, determining what operations need to be performed on the data, and deciding on the order in which to execute the operations. The planner must take into account the cost of accessing the different information sources, the cost of retrieving intermediate results, and the cost of combining these intermediate results to produce the final results. A partial-order parallel-execution planner is ideally suited for this problem since the parallelization is for efficiency purposes, and there are many possible plans for retrieving the same data and the choice of plans is crucial in determining the overall efficiency.

Figure 5 shows an example parallel query-access plan. The three basic query access planning operations used in this plan are move-data, join, and retrieve. The move-data operation moves a set of data from one information source to another. The join operation combines two sets of data into a combined set using the given join relations. The retrieve operation specifies the data that is to be retrieved from a particular information source.

The domain and planner described here are fully implemented and serve as an integral part of an information retrieval agent. We have also extended UCPOP to perform execution, and to do so in parallel. The system is implemented and runs in Lucid Common Lisp on SUN and HP workstations. To provide a sense for the potential speed-up of this approach we ran a sample query that involved queries to two different databases. Without parallelization, the system generated a plan with six operators in 0.82 CPU seconds and then executed the plan in 101.8 seconds of elapsed time. With parallelization, it generated the plan in 1.3 CPU seconds and executed the plan in 62.4 seconds of elapsed time, a 39 percent reduction in execution time.

Related Work

An alternative approach to addressing the problem of simultaneous execution is provided by work on temporal planning (Allen et al. 1991; Penberthy 1993). A temporal planner can handle the general problem of simultaneous parallel execution, but this general solution has a cost, since just testing the satisfiability of a set of assertions is NP-hard (Vilain, Kautz, & van Beek 1989). The capabilities of a full-blown temporal planner are necessary only if we need to explicitly reason about the interaction between parallel actions. In this paper we focus on the simpler problem of non-interacting simultaneous execution, which does not require a full-blown temporal reasoner to handle. In fact, partial-order planners appear to be well suited for problems in this class.

Another approach to this problem is to generate totally-ordered plans and then convert each plan into a partially-ordered plan (Veloso, Perez, & Carbonell 1990; Regnier & Fade 1991). The problem with this approach is that the particular choice of the totally-ordered plan determines the parallel execution plan. As such, in order to consider the space of parallel execution plans requires searching through the space of totally-ordered plans. Since a single partially-ordered plan often corresponds to a number of totally-ordered plans, it will be harder to efficiently search the space of parallel execution plans.

Recently, Backstrom (1993) showed that the general problem of finding an optimal parallel execution plan
is NP-hard. We cannot escape from this complexity result; however, partial-order planners do avoid the NP-hard subproblem of testing satisfiability and provide a more natural framework than total-order planners for searching the space of parallel plans and encoding domain-specific control knowledge to guide the search.

Discussion

The idea of using partial-order planning to generate parallel execution plans has been around since the early days of planning. What we have done in this paper is to explicate the underlying assumptions and situations where parallel execution is possible, characterize the differences in the plans produced by various planning algorithms, and identify the changes required to use UCPOP as a parallel execution planner. We have also shown that these ideas apply directly to the problem of generating parallel query access plans.

In future work we plan to tightly integrate the planning and execution components. This would allow the system to dynamically replan actions that fail, while continuing to execute other actions that are already in progress. In addition, we plan explore the problem of how to efficiently search the space of parallel execution plans. First, we will consider domain-independent search strategies that produce the highest quality solution that can be found within the time allotted. Second, we will exploit domain-specific knowledge to both restrict the search space and guide the search.

Acknowledgments

I would like to thank Oren Etzioni, Rao Kambhampati, Kevin Knight, Milind Tambe, and Dan Weld for their comments and suggestions on earlier drafts of this paper. I would also like to thank Chin Chee for his help in running the experiments, and Don McKay, Jon Pastor, and Robin McEntire at Unisys for their help in setting up the remote database servers, which made it possible to run parallel queries.

References


