Adaptable Planner Primitives for Real-World Robotic Applications

Robert W. Wisniewski and Christopher M. Brown*
Department of Computer Science
University of Rochester
Rochester, NY 14627-0226
voice: 716-275-0469 fax: 716-461-2018
{bob and brown}Dcs .rochester.edu

Abstract

With increased processor speed and improved robotic and AI technology, researchers are beginning to design programs that can behave intelligently *and* interact in the real world. A large increase in processing power has come from parallel machines, but taking advantage of this power is challenging. In this paper we address the issues in designing planners for real-time AI and robotic applications, and provide guiding principles. These principles were designed to minimize the difference between the new real-time model and the standard off-line model. Applying these principles yields a better-structured application, easier design and implementation, and improved performance. The focus of the paper is on a design methodology for implementing effective planners in real-world applications. Using Ephor (our runtime environment), and applying the described planner principles, we demonstrate improved performance in a real-world shepherding application.

1 Introduction

With increasing processor power there has been growing interest and research work in designing intelligent applications that interact with the real world. Combining real-world requirements with uncertain and complex cognitive activities leads to issues of resource allocation and decision-making in real time not previously encountered in AI or robotics applications. We call intelligent parallel applications with unpredictable aspects and a complex mixture of competing and cooperating demands Soft PArallel Real-Time Applications, or *SPARTA s.* Building SPARTAs is difficult because it involves not only designing the intelligent portion of the application, but also handling both soft real-time constraints (e.g. robot planning), and hard real-time constraints (e.g. a balance subsystem).

If a SPARTA programmer is oblivious to real-time issues when designing an application, poor or incorrect behavior may result. Tension arises due to the discrepancy between how an AI programmer wants to design an application and the requirements of planning and acting in the real world. We have developed Ephor[1], a runtime environment[2] to support SPARTA development and execution. Our goal is to provide as much of the off-line programming model as possible, so that the standard techniques available for designing intelligent robotic applications can be applied to SPARTAs.

In this paper we focus on techniques for implementing effective planners[3] in parallel real-world applications. Previously, designing a planner for a SPARTA meant tracking resource allocation, timing tasks, and handling other concerns of interacting in the real world. The combination of Ephor and our model of planning in SPARTAs considerably simplies design.

A key observation we will leverage throughout our discussion is that in a dynamic real-world environment it is important to be able to adapt. While this may be intuitive, its implications for planner (and runtime) design are significant. The importance of adapting holds both for the action taken by the application as well as how the application decides on that action. More concretely, in later sections we discuss the advantages of having several planners (with the same *goal)* varying in resources consumed (and thus quality of result). This diversity is useful because it allows an adaptive decision to be made during execution when the application needs a particular *goal* solved. In part, the principles for designing SPARTA planners are motivated by what tools/mechanisms the underlying runtime environment

*This material is based upon work supported by NSF Research Grant number CDA-8822724, DARPA Research Grant MDA972-92-J-1012, and ONR Research Grant number N00014-93-1-0221. Robert Wisniewski was supported in part by an ARPA Fellowship in High Performance Computing administered by the Institute for Advance Computer Studies, University of Maryland. The Government has certain rights in this material

[1] Ephor WAS the name of the council of five in ancient Greece that effectively ran Sparta.

[2] A runtime environment is a combination of library calls and system code working together for a specific purpose, e.g., a lisp interpreter can be thought of as a runtime environment.

[3] Throughout, "planning" refers to all forms of cognitive reasoning, problem-solving, and decision-making techniques for deciding what to do next, from simple random choice through sophisticated modern planners.

and operating system can provide to the application programmer. Creating a happy marriage between what can be supported (from the system's point of view) and the ideal AI programmers' model, is extremely important to implementing successful robotic real-world applications.

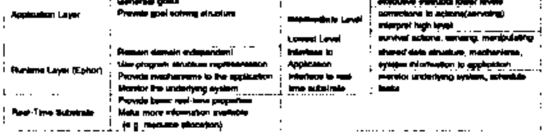
Throughout, we use the specific application domain of shepherding to provide concrete examples of our principles, and to demonstrate their effectiveness. The shepherding application domain is flexible and maps onto a large class of real-world AI applications that involve uncertain actions, uncertain sensing, real-time constraints and responsibilities, planning and replanning, dynamic resource management, dynamic focus of attention, low-level reflexive behaviors, and parallel underlying hardware (e.g. purposive vision, autonomous vehicle control and navigation). A real-world shepherding implementation runs in our robotics laboratory [Ballard and Brown, 1992][von Kaenel and Wisniewski, 1994](see Figure 1), but the results in Section 4 are from a real-time simulator that allows greater flexibility in experimentation. The implementation consists of self-propelled Lego vehicles ("sheep") that move around the table ("field") in straight lines but random directions. Each sheep moves at constant velocity until herded by the robot arm ("shepherd"), which redirects it towards the center of the field. A second robot arm ("wolf") can encroach on the field and remove ("kill") sheep if not prevented. The shepherd has a finite speed and can affect only one sheep at a time. The goal of the shepherd is to keep as many sheep on the table as possible, and the more powerful the sheep behavior-models and look-ahead, the better the results.

Figure 1: The Real-World Shepherding Application (camera overhead)

General approaches to designing SPARTAs are only now beginning to emerge, and usually individual solutions do not generalize well. We believe this is a two part problem. First, underlying support for developing a general framework is needed. Secondly, the principles in designing effective planners for these types of applications must be understood. In other work we have addressed the first concern. In this paper we address the second by providing principles for SPARTA planner design that yield a better structured application, simplify design, and improve performance. We start by providing a model of SPARTAs in Section 2. Section 3 lists and describes in detail the principles involved in designing effective planners for SPARTAs. This section is the focus of the paper. In Section 4 we provide a quick overview of our application and support, and results demonstrating the effectiveness of the planner principles. Section 5 provides conclusions and describes continuing work.

2 A Model of SPARTAs

This section is devoted to discussing models for designing SPARTAs. We describe the model we have chosen and therefore its applicable domains. There are two disparate approaches to designing real-world AI and robotic applications. A subsumption model [Brooks, 1987] [Brooks, 1989] claims intelligent behavior will emerge from low-level reactive modules. While our model includes reactive modules as part of its real-time component, the allowance for time-constrained high-level reasoning places it in the second, more traditional AI camp. As in a modular architecture [Fodor, 1985], we assume different, loosely coupled mechanisms for low-level reaction-perception and high-level reasoning.



Real-World Layer	Responsibilities	Layer Division	Characteristics
Application Layer	Respond to environment Generate goals Provide goal solving structure	Executive Level	agent determines next action, search executive instructs lower levels
		Intermediate Level	corrections to actions(servoing) interpret high level
		Lowest Level	survival actions, sensing, manipulating
Runtime Layer (Ephor)	Remain domain independent Use program structure representation Provide mechanisms to the application Monitor the underlying system	Interface to Application Interface to real-time substrate	shared data structure, mechanisms, system information to application monitor underlying system, schedule tasks
Real-Time Substrate	Provide basic real-time properties Make more information available (e.g. resource allocation)		

Figure 2: The Three Layers in the Design of a Real-World Application

We augment the cognitive-reactive dichotomy with an intermediate, run-time layer (Figure 2). All the application levels reside above the the runtime layer and real-time substrate. The application interacts only with Ephor and not directly with the substrate. At the lowest [application] level are [hard] real-time periodic or aperiodic (environment responsive) tasks. These tasks will almost require the same set of resources and run for predicatable amounts of time.

The intermediate [application] layer serves several functions. Among them, it "catches the mistakes" of the lower level and "interprets the meaning" of the higher layer. The former involves servoing, adjusting sensors and manipulators to ensure the intended action is actually carried out by the lower level (e.g. servo robot arm to sheep). The latter involves parsing a high level description into components that can be understood and executed by the lower level (e.g. determine robot arm instructions to herd "that" sheep "there").

The highest or executive level consists of planning, reasoning, information gathering and processing, decision analysis, etc.. With a traditional run-time and underlying operating system, the most significant differences between applications occur at the executive level. Unfortunately it is this level that is most task-dependent and has the fewest standard formalisms. Researchers have approached this aspect of SPARTAs from different angles. Both [Gervasio and DeJong, 1992] and [Hadavi *et al.*, 1992] describe planners effective in handling varying environmental factors. Hoogeboom and Halang [Hoogeboom and Halang, 1992] propose a more general approach suggesting that "In anticipation of a deadline at which some task must be fulfilled, it should be possible to choose from different program segments the one that maintains optimum performance." We concur.

A goal in our work, and implicit in the planner specifications, is the desire to design a general architecture rather than just one for a specific application. While there are accounts of specific applications [Brown and Terzopoulos, 1994] that have clear design principles and correct behavior, it is difficult to extract useful code from these programs to help design another SPARTA. Some other work that, has looked at dynamic tradeoff decisions is Schwuttke and Gasser's [Schwuttke and Gasser, 1992] dynamic trade-off evaluation algorithm that decide which data to monitor in a spacecraft. Durfee [Durfee, 1990] suggests a more general method of supporting individual cooperating components. Other prominent work in this area of developing general mechanisms for supporting SPARTAs is by Gopinath and Schwan [Gopinath and Schwan, 1989] who suggest objects that can move along a continuum of resource use and describe mechanisms for scheduling these objects in a distributed system.

3 Planner Design and Necessary Support

3.1 Background

Let a *goal* be something the application wishes to accomplish (e.g., save sheep) and a *technique* be a method or algorithm for accomplishing a goal (e.g., planner A). Figure 3 illustrates the model of the program structure for a SPARTA. Throughout a program's execution it will need to solve many goals and frequently will need to solve the same goal repeatedly. If each goal has only a single, sequential, fixed technique to solve it, then there will be no flexibility in choosing a technique for solving a goal and thus the program will have sacrificed an entire dimension of adaptability (it can only choose different goals indicating a different course of action).

Interacting with the real world implies coping with the unknown and the uncertain. Goals may be generated in regards to unexpected environmental stimuli. As a specific example from our real-world shepherding application consider the entry of a wolf into the field; a high priority goal ("kill wolf) must be solved. Some goals may take longer or shorter than expected because of a change in the environment or because of varying amounts of available resources (if while solving the "find next sheep to save" goal on seven processors, six of them are preempted for other tasks, this goal will take consider-

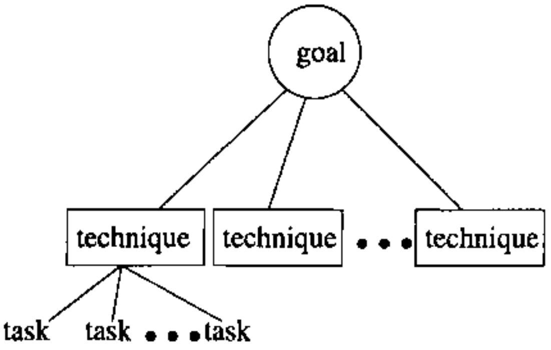


Figure 3: Program Structure

ably longer than originally expected to solve). An unpredictable environment can also cause additional goals to be needed while no longer requiring the results of others. For example, if while in the middle of solving the "save sheep goal", the robot arm is allocated to killing a wolf, there is no reason to continue to solve the "save sheep goal" since the robot arm will not be available, instead the processor(s) could be freed and given to another goal. Clearly, the internal state of a SPARTA application, runtime, and operating system will be highly variable over time. To make efficient use of resources the application must cooperate with the underlying runtime to allow the whole system to adapt dynamically to varying conditions. Using worst case analysis to pre-configure the system is too inefficient [Paul *et al.*, 199l][Strosnider and Paul, 1994].

With our model, the SPARTA programmer can still conceptualize their program in terms of goals that need to be solved and techniques for implementing those goals. The only difference between previous models for off-line applications and the model described by our planner principles for SPARTAs is the emphasis on specifying several ways of solving a given goal. Of course, to take advantage of this new model, underlying support is required (see [Wisniewski and Brown, 1993]), and the ability to inform the underlying support about the goals and techniques. We have developed a simple scheme that allows the programmer to communicate a SPARTA's program structure to the underlying runtime environment, which we briefly describe in Section 4. Convinced that we can support such a model we devote the rest of this section to discussing the details and giving examples of the our planner principles.

3.2 Designing a Suite of Planners

It is essential for SPARTAs to maintain as much flexibility as possible both in their ability to choose different courses of action based on the environment and their ability to have multiple ways (techniques) for *determining* a particular course of action. Below we list (in order of increasing effectiveness) a set of principles for planner design. After the list we describe each item in detail and provide examples. The more of the principles that are followed when designing a SPARTA planner the better the program's performance will be.

1. Provide techniques that can vary in completion time (e.g., anytime algorithms [Liu *et* a/., 1991]).
2. Provide multiple techniques that:
   a) use different resources (e.g., infrared sensor/binocular vision)
   b) vary (significantly) in quantity of resources used (cpu time, etc.).
3. Provide parallel planners that:
   a) use a "bag of tasks" (processor farm) model
   b) use different resources
   c) vary in quantity of resources used.

The intent of these recommendations is to provide flexibility of resource allocation in as many dimensions as possible. The more flexibility designed at this level the more adaptable the program will be to unforeseen events since the underlying runtime environment (Ephor) will be able dynamically to select from a more diverse set of techniques and thus will more likely be able to find an appropriate technique for a given situation.

3.3 Description of Planner Principles

Following the outlined planner principles yields an adaptable program that the allows the runtime to adjust to unexpected events and thus achieve increased performance. For each principle above we provide a detailed description and give an example from the shepherding domain described in the introduction.

1) Provide techniques that can vary in completion time

A method for meeting the challenge of time variability is to design a planner that can move along a continuum of completion times as suggested by Gopinath and Schwan [Gopinath and Schwan, 1989]. Such a technique is similar to the motivation behind anytime algorithms or imprecise computations [Liu *et* a/., 199l] in which after a certain minimum time the program's result improves until a final completion time. These methods allow the underlying system dynamically to allocate the maximum amount of time available to the goal while still allowing for early termination if processor cycles are needed by other goals.

An example from our shepherding application is the vision processing goal of determining the centroid of each (circularly marked) sheep. A quick approximation of the centroid may be found by scanning a horizontal and then vertical line [von Kaenel and Wisniewski, 1994]. After this initial phase we have a reasonable centroid. Continuing by searching every pixel and using a weighted mean to determine will provide more accurate results, and given time, would be preferred.

2a) Provide multiple techniques that use different resources to achieve the same goal

Ideally, these techniques would have non-intersecting resources, but techniques using different but not unique sets are still useful. For example, having one technique that uses processing time, a camera, and the robot arm, and another that uses processing time, an infrared sensor, and the robot arm. During execution, when this goal needs to be solved, the runtime will still be able to solve the goal even if the infrared sensor is allocated to another technique by running the technique that uses the camera.

As a specific example consider a mobile robot that has two techniques it can use to find the distance to a wall. It has an infrared sensor that may provide a fast response and a pair of binocular cameras it can also use. If both resources are free when it needs to solve the get-distance-to-wall goal, then it prefers to use the infrared sensor because it is faster. However, if the high-level executive decides it is time to obtain the distance to the wall when the infrared sensors are being used to avoid an object, the runtime can still solve the goal by running the technique that uses binocular vision.

2b) Provide multiple techniques that vary in quantity of resources used

Make the techniques differ in the amount of resources they use and consequently the quality of the result they produce. The most straightforward example is the amount cpu time consumed. Anytime algorithms capture this notion and are supported by Ephor, but even more significant differences yield greater adaptability. For example: emergency or reflexive 0(1) algorithms, heuristic $O(n^{2 or 3})$ algorithms, or brute force search $0(2^n)$ algorithms. Concentrate on designing tasks within constant factors of the expected time available for this task. This dimension of flexibility allows the runtime dynamically to select the best technique based on the internal load on the SPARTA's resources. There may be periods of time when the application desires many goals to be solved simultaneously and other periods of relative inactivity. We have found a diversity of planners provides the best overall behavior for a given goal, because during quiet periods a higher quality technique can be run and during periods of high demand a simple technique can still be run (as opposed to being unable to run any technique).

For example, in the shepherding application we have implemented a simple planner that just looks for the first sheep it finds moving away from the center and computes the intercept to save it. We also have implemented three variations of another planner varying in the amount of lookahead performed. One looks two sheep saves into the future when considering a move. The others look three and four moves. Lookahead is useful because, for example, it may be the case that by letting the farthest sheep from the center go and moving to the other side of the field two sheep can be saved. These different planners are extremely valuable because they provide alternatives to Ephor. We provide results in Section 4 that show with a variety of planners, we can achieve better behavior (more sheep confined) in the shepherding application. We have also implemented a depth n search (where n is the number of sheep), but in practice it never has enough time to run for n > 4.

3a Provide parallel planners that use a "bag of tasks" model

As mentioned in the introduction, SPARTAs contain parallel and distributed components that bring a new level of complexity and a new set of issues to designing real-world applications. However, this parallelism also brings new opportunities for adaptation. There are many models of parallel computation (e.g. data parallelism is natural in low-level vision). Programming a technique to have a fixed number of tasks on a fixed set of processors is counterproductive since it does not allow for any adapting. Instead, a model of parallelism is needed that can easily and quickly change in light of varying and unpredicted environmental stimuli. Our experiments show that if the application is programmed with a "bag of tasks" model, the runtime can provide considerably better performance. In a "bag of tasks" model, work is divided up into reasonable size pieces [of the problem] and placed in a central repository. Each process removes a piece from the bag, processes it, and possibly updates shared information with the result. Examples of this model of parallelism are the Uniform System [Thomas and Crowther, 1988] or the Problem-Heap Paradigm [Cok, 1991; Moller-Nielsen and Staunstrup, 1987]. This paradigm provides tremendous flexibility since the runtime can choose to run any number processes to work on this technique.

An example from the shepherding application is a parallel planner we designed. This planner looks at the next n (for our experiment it was four) possible sheep saves in order to determine the best next move. Another way to cast the planning problem is to look at all permutations of the sheep in the field and count the number of sheep still confined at the end of the sequence of sheep saves and the amount of time taken to do so. This representation nicely fits the "bag of tasks" model since now we put into the bag a set of all the possible permutations. Each (identical) process pulls a permutation out of the bag computes the information above and updates a central location (holding the best permutation seen so far) if it determines it has found the best option so far. We give results from applying the "bag of tasks" model to the shepherding domain in Section 4.

3 Provide Parallel planners that: b) use different resources c) vary in quantity of resources used.

Design parallel planners that either use distinct resources or that vary greatly in the quantity of resources consumed. The arguments and benefits are analogous to those we discussed for sequential planners in 2a and 2b.

4 Supporting Results

In this section we examine the effectiveness of our planner principles. The evaluation is accomplished by using Ephor (our runtime environment), and real-time primitives from IRIX (the operating systems of our 12 processor SGI Challenge). First, we briefly explain the application-Ephor interface, which is how the application informs Ephor of its goals, techniques, and task. Then, we provide results demonstrating improved performance using adaptable planner principles. We concentrate on two specific principles. We show that application behavior improves when multiple techniques accomplishing the same goal are available to the runtime and when a parallel planner it used that can adapt the number of processes.

```
vision_proc_goal * ephor_create_goal(ephor_periodic,
            True, ephor_priority, 1,
            ephor_goal_name, "vision proc",
            ephor.rate, 16666, NULL);  /* 60 HZ */

temp_tech = ephor_create_technique(vision_proc_goal,
            ephor_cpu_time, 5000, NULL);

vision_proc_id ■ ephor_create_task(vision_proc_goal,
            temp.tech, ephor.imp_function,
            vision_processor, NULL);
```

Figure 4: Application-Ephor Interface

At startup, the application informs Ephor of its goals, the different techniques it has for solving those goals, and the specific function(s) that implement the techniques. Throughout the programs' execution, when it wishes to solve a goal it informs Ephor. The application can also specify periodic goals. Once Ephor knows about the application's goals, techniques, and tasks, it dynamically selects the most appropriate technique when the application informs Ephor it wants to solve a goal.

As a simple example from the shepherding application, Figure 4 shows how the vision processing goal is defined. The first call ephor_create_goal returns a handle to the vision processing goal. If this goal was not periodic, i.e., called in response to the environment, then ephor_periodic would be False, and the program would call ephor_run_goal(vision_proc_goal) when it wanted this goal run.

4.1 A Suite of Techniques

The first planner principle we evaluate is the usefulness of generating different techniques for accomplishing the same goal. Here we will examine the performance of two planners (planner A and B) - a more thorough analysis of these tradeoffs may be found in [Wisniewski and Brown, 1993]. Both these planners figure out the next sheep to save but differ in how long they take to run and how many sheep are contained (in steady state) when running on an unloaded cpu. To guarantee accurate measurement (no competing load) for this experiment, we dedicate one processor to the planner function. A sheep can travel from the center to the edge of the table in 10 seconds and the shepherd in about 1/3 of a second.

Planner A computes a list of all the sheep moving away from the table center that the shepherd has time to reach, sorted by distance from the center. It then determines the best order for saving the next four sheep: this requires future prediction of sheep movements. The best sequence is the one maximizing the number of sheep saved. Among the orderings that save equal number of sheep, preference is given to the ordering taking less time. The first sheep in the sequence is saved and the planner starts over. Planner A performs the best under no load but takes the longer time to run (about one second). Planner B is a reactive, no look-ahead strategy, that simply tries to save the sheep farthest from the center. It runs much faster than A (about 8 milliseconds) but does not perform nearly as well under no load: if by letting the farthest sheep go, it is possible to save the next two and otherwise not, A will save the two sheep while B will save only one.

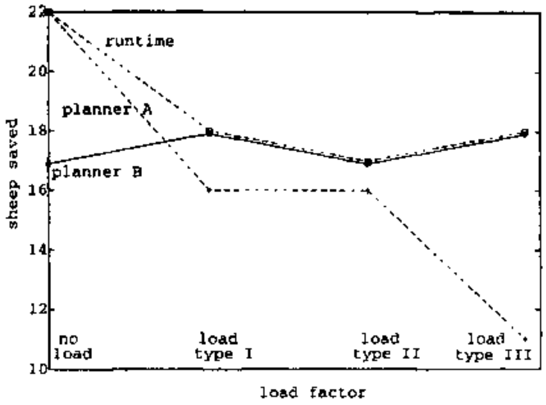


Figure 5: Adapting to high fixed loads

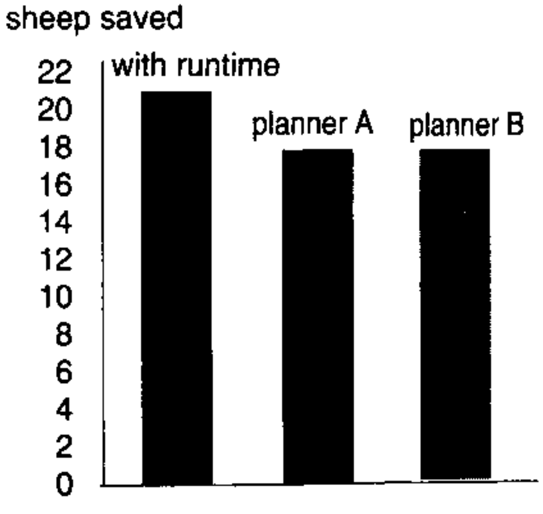


Figure 6: Adapting to high variable loads

To compare the different planners under simulated conditions of parallel activity in other parts of the SPARTA, a controlled load was placed on the processor that the planner was running on. Since we were using a multiprocessor we could vary the load experienced by the planner process without affecting any of the other processes in the system. We also had tight control over how much load was experienced on the processor running the planners. The line graph of Figure 5 (on the left) gives performance for a set of fixed loads (the load does not vary throughout the entire execution - an unrealistic model since in a real application goals will come and go, but it allows us to see the comparative benefits of each of the planners), while the bar chart of Figure 6 gives average performance when the cpu load varies during the run (like actual program execution).

The experiment (Figures 5 and 6) demonstrates the effect of a high load. Recall that B runs about 40 times faster than A. Planner A is expected to outperform B with no load, but under increased load planner A might not complete its calculations in time, thus planner B is expected to outperform A under high load. The loads are plotted on a logarithmic scale: load type II is twice as much background load as load type I and half as much as load type III. Indeed there is a dramatic decrease in performance of planner A under higher loads while planner B remains fairly constant. In a fixed load environment the run-time can select the better of the two planners, thus achieving the best performance in all cases. In the second half of the experiment the load varied through time. Half the time there was no load and half the time there was load. When there was a load it was divided evenly (by thirds) amongst the different load types. Figure 6 represents how A, B, and the run-time mixture perform under varying load; best performance occurs when Ephor dynamically selects the planner to suit the (currently) available resources. It is clear that dynamically selecting between planners improves application performance.

4.2 Adaptable Parallel Planners

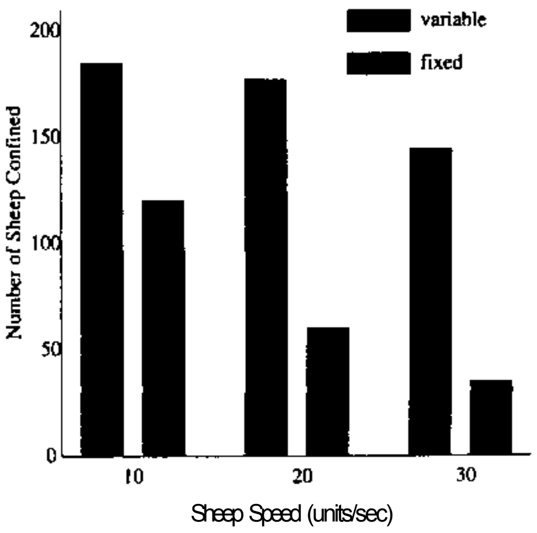


Figure 7: Fixed versus Adaptable Parallel Planners

The second planner principles we evaluate is the "bag of tasks" (see section 3a) model for parallel planner design. As we have mentioned, a key aspect of performing well in the real world is being able to adapt. This adaptability applies both to the runtime and the application. Our principle of using a "bag of tasks" model is motivated by the fact that it provides considerable flexibility when considering the amount of processing power to allocate to a planner.

In this experiment we again assume a model of varying load as would be observed in a real application. The parallel planner has been written using a "bag of tasks" model. The planner looks four saves into the future. The possible permutations for the next four sheep saves are placed into a central queue where they are removed by as many processes as the runtime has decided to invoke in this particular goal instantiation. This planner is qualitatively different from the planners discussed in section 4.1 and the environment is quantitatively different, so the results should not be compared.

The results appear in Figure 7. The load of the application allowed for between 1 and 7 processors to be available to the save-sheep goal at different points throughout its execution. The "fixed" bar represents the application's behavior assuming it could not adapt and adjust to use the extra processors available at various points throughout the execution, i.e., it always used one processor. This would be required if the planner could not adapt to the number of processors it could use, otherwise it would not have been able to run in times of slightly heavier load. The variable bar represents when Ephor was allowed dynamically to allocate more processors for the parallel planner when they were available. We vary the response time expected from the planner by varying the rate the sheep move. Notice that when the response demands placed on the application increase, it becomes more important for the planner to have been designed allowing for differing number of processes to be used. This figure illustrates the benefit of using a flexible parallel planner. When the sheep are moving around quickly, the adaptable parallel planner can confine over three times as many sheep as the fixed parallel planner. The important aspect of this discussion is not that parallel planners (versus sequential ones) can improve the performance of applications, rather that in real-world applications we need planners that can dynamically vary the number of processors they use. And equally important, we need a system such as Ephor, that can support this desired behavior.

This graph represents the interesting part of the state space for the save-sheep planner. If the sheep move very slowly (1-5 units/sec) then either planner will have enough time to do something reasonable, and similarly if the sheep are moving extremely quickly (greater than 60 units/sec) neither planner will have enough time *to* do anything. Achieving perfect speedup when running on seven processors (versus one) would allow the planner to run seven times as fast. This does not however, translate into a seven-fold improvement in application behavior. In fact there may be some programs that do not benefit or benefit very little from decreased running time. However, many real-world applications will be able to benefit from being able to perform more computation in less time. Exploiting the parallel dimension of programming a SPARTA can substantially improve application behavior.

5 Conclusion

We described SPARTA characteristics and set of general design principles for planners and parallel planners in real-world applications. We presented qualitative arguments for some, and for others we presented results indicating significant performance improvement (as much as 300 percent on our system). In all cases we showed the key to good (parallel) planner design in SPARTAs was the ability to adapt, by offering a set of techniques to solve a particular goal.

We have found it very productive to work on both the high level application and runtime system simultaneously. Their interactions are nontrivial and hard to anticipate. Our current and future emphasis is on investigating and expanding the runtime support for sophisticated parallel planners and decision-makers. We are also continuing to develop better measures for how well the implemented support meets the need of the high-level planner.

References

[Ballard and Brown, 1992] DTI. Ballard and C. M. Brown. Principles of animate vision, *cvgip,* 56(1):3-21, July 1992.

[Brooks, 1987] Rodney A. Brooks. Intelligence without, representation. In *Proceedings of the Workshop on the Foundations of Artificial Intelligence,* 1987.

[Brooks, 1989] Rodney A. Brooks. A robot that walks: Emergent behavior from a carefully evolved network. *Neural Computation,* l(2):253-262, 1989.

[Brown and Terzopoulos, 1994] Christopher M. Brown and Demetri Terzopoulos. *Real-Time Computer Vision,* chapter Chapter 9: Hybrid problems need hybrid solutions? Tracking and controlling toy cars, pages 209-230. Cambridge University Press, 1994.

[Cok, 1991] Ronald S. Cok. *Parallel Programs for the Transputer.* Prentice Hall, 1991.

[Durfee, 1990] Edmund H. Durfee. Towards intelligent real-time cooperative systems. In James Hendler, editor, *Planning in Uncertain, Unpredictable, of Changing Environments: 1990 AAAI Spring Symposium,* 1990.

[Fodor, 1985] Jerry A. Fodor. Precis of the modularity of mind. *The Behavioral and Brain Sciences,* 8(1): 1-5, 1985.

[Gervasio and DeJong, 1992] Melinda T. Gervasio and Gerald F. DeJong. Completable scheduling: An integrated approach to planning and scheduling. In *AAAI Spring Symposium on Practical Approaches to Scheduling and Planning,* pages 122-126, Stanford University, March 1992.

[Gopinath and Schwan, 1989] Prabha Gopinath and Karsten Schwan. Chaos: Why one cannot have only an operating system for real-time applications. Operating Systems Review, 23(3): 106-125, July 1989.

[Hadavi et al, 1992] Khosrow Hadavi, Wen-Ling Hsu, and Michael Pinedo. Adaptive planning for applications with dynamic objectives. In AAAI Spring Symposium on Practical Approaches to Scheduling and Planning, pages 30-31, Stanford University, March 1992.

[Hoogeboom and Halang, 1992] Boudewijn Hoogeboom and Wolfgang A. Halang. Real-Time Systems Engineering and Applications, chapter 2: The Concept of Time in the Specification of real-Time Systems, pages 11-40. Kluwer Academic Publishers, 1992.

[Liu et al, 1991] Jan Liu, Kwei-Jay Lin, Wei-Kuan Shih, Albert Yu, Jen-Yao Chung, and Wei Zhao. Algorithms for scheduling imprecise computations. IEEE Computer, 24(5):58-69, May 1991.

[Moller-Nielsen and Staunstrup, 1987] A P. Moller-Nielsen and A J. Staunstrup. Problem-heap: A paradigm for multiprocessor algorithms. Parallel Computing, 4:64-74, 1987.

[Paul et a/., 1991] C. J. Paul, Anurag Acharya, Bryan Black, and Jay K. Strosnider. Reducing problem-solving variance to improve predictability. Communications of the ACM, 34(8):80-93, August 1991.

[Schwuttke and Gasser, 1992] U. M. Schwuttke and L. Gasser. Real-time metareasoning with dynamic trade-off evaluation. In Proceedings of the Tenth National Conference on Artificial Intelligence AAA! 1992, pages 500-506, 1992.

[Strosnider and Paul, 1994] Jay K. Strosnider and C. J. Paul. A structured view of real-time problem solving. AI Magazine, 15(2):45—66, summer 1994.

[Thomas and Crowther, 1988] R.H. Thomas and A W. Crowther. The uniform system: An approach to run-time support for large scale shared memory parallel processors. In Proceedings of the 1988 International Conference on Parallel Processing, pages 245-254, St. Charles IL, August 1988.

[von Kaenel and Wisniewski, 1994] Peter von Kaenel and Robert W. Wisniewski. Real-world shephering - combining vision, manipulation, and planning in real time. Technical Report 530, Department of Computer Science, University of Rochester, Rochester, NY, August 1994.

[Wisniewski and Brown, 1993] Robert W. Wisniewski and Christopher M. Brown. Ephor, a run-time environment for parallel intelligent applications. In Proceedings of The IEEE Workshop on Parallel and Distributed Real-Time Systems, pages 51-60, Newport Beach, California, April 13-15, 1993.