Options for Parallelizing CASPER for a Mesh Processor

Bradley J. Clement, Tara A. Estlin, Benjamin D. Bornstein

Jet Propulsion Laboratory, California Institute of Technology
4800 Oak Grove Drive, M/S 301-260, Pasadena, CA, 91109
FirstName.MiddleInitial.LastName@jpl.nasa.gov

Abstract

Space missions have a growing interest in putting multi-core processors onboard spacecraft. For many missions processing power significantly slows operations. We investigate how continual planning and scheduling algorithms can exploit multi-core processing and outline different potential design decisions for a parallelized planning architecture. We focus on options for adapting an existing planner implemented in C++. This organization of choices and challenges helps us with an initial design for parallelizing the CASPER planning system for a mesh multi-core processor. This work extends that presented at another workshop with some preliminary results.

Introduction

While processor speeds are leveling off, chip manufacturers are still keeping pace with Moore's Law in numbers of transistors and are producing processors with increasing numbers of CPU cores. Space flight software could particularly benefit from these advances because of the limited computation of current flight processors, the many applications competing for CPU time, the potential for lower power computation (Kogge et al., 2010), and the flexibility to power-off a subset of processors and to separate and duplicate processes for fault tolerance. Future space missions may take advantage of parallel computing hardware, including many core processors, such as Tilera's 64 core TILE64™ (Wentzlaff et al., 2007), or field-programmable logic devices, such as Xilinx's Virtex-5™.

For example, image processing often requires a relatively large amount of computation time and can be a bottleneck in operations. Mars Exploration Rover operations are slowed while waiting on visual odometry and obstacle avoidance software that rely on image processing and cannot keep up with rover movement. In addition, faster visual odometry might have helped prevent the Spirit Mars Exploration Rover from getting stuck in its current sand trap and would have been beneficial during the attempted extrication (November - February 2010).

Not only can missions benefit from faster computation but also from new capabilities that additional processing capability enables. Planning and scheduling software can take advantage of extra processing power to refine higher-level goals into command sequences, verify the safety of those commands, and alter the planned sequence based on unexpected events. This capability can simplify ground operations, but more importantly, having the capability onboard gives the spacecraft greater flexibility and robustness to carry out longer sequences without human assistance. This greater autonomy can be even more beneficial or required for deep space missions where communication is greatly delayed by the light-time to travel long distances or when communication is not possible because the spacecraft is on the far side of a planet or the Sun.

Although not specifically aimed at parallelization, GEMPLAN and its next generation implementation, COLLAGE, divide planning problems into loosely coupled sub-problems (Lansky, 1991). The parallelization of planning and scheduling has been investigated indirectly through distributed planning and scheduling (e.g., Georgeff, 1983; Ephrati and Rosenschein 1994; Wolverton and desJardins, 1998; Clement and Barrett, 2003; Brafman and Domshlak, 2009). These approaches typically describe how agents can solve sub-problems locally and coordinate over shared parts of the environment. Distributed constraint satisfaction and optimization algorithms have also been applied to scheduling, giving each variable or group of variables its own decision-making process (e.g., Sycara et al., 1991; Maheswaran et al., 2004). Recent, direct approaches to parallelization of planning divide the search space among processors, typically by load balancing search state expansion (e.g. Zhou and Hansen, 2007; Burns et al., 2009; Kishimoto et al., 2009).

The choice of how to parallelize algorithm computation can be especially difficult because of the number of options to consider. Should the problem be divided and how? What are the bottlenecks in computation, and how can they be parallelized? How are shared constraints coordinated? How are the local solutions merged? Should there be threads or separate processes? Should information be communicated using shared memory or messages? What middleware should be used? fork? Pthreads? MPI? Should distributed planning software be considered?

Many parallelization decisions and challenges are based on hardware architecture. For example, each core of the TILE64™ processor has its own L1 and L2 caches, which can be shared with other cores, forming a unified L3 cache

Planning Data	Planning functions
plans/schedules	plan and schedule: identify flaws, add/delete search states

• **activities**
  ○ **parameters, possible values**
  ○ **parameter dependencies**
  ○ **parameter/state constraints**
  ○ **reservations**
  ○ **temporal constraints**
  ▪ **valid time intervals/orderings**
	add, delete, constrain, move, detail, abstract
	get, set, choose value
	evaluate dependency function, propagate values, check if stale
	find valid values, check if violated, propagate constraints
	apply to state/resource timelines
	add, remove
	compute valid time intervals/orderings

• **future state/resource variables (timelines)** ○ **values**	compute valid time intervals, identify flaws compute, propagate, get contributing activities

• **constraint rules**
  ○ **flaws**
• **preference/optimization criteria**
  ○ **scores, deficiencies**
	identify flaws
	choose flaw, choose resolution method (e.g. move, add delete)
	compute scores, identify deficiencies
	choose preference to improve

Table 1. The basic data and associated functions a planning/scheduling system may implement. Highlighted functions are common bottlenecks for CASPER domains.

(Ungar and Adams, 2009). Because memory access is chained through adjacent processors, the time required to access a memory location in the L3 cache is roughly proportional to the Manhattan distance from the processor to the cache where it resides. In addition, the processor is limited to four memory controllers for accessing the RAM, so memory access from one core may be waiting on those of other cores. Therefore, parallelization will benefit from keeping data close to the processors that need it. One strategy is to use the cores located next to the memory controllers to stream data to the others, possibly for pipelining data processing across adjacent cores.

Here we describe our approach to exploring this space of options for the CASPER planning and scheduling system (Chien at al., 2000b). We outline the space of design choices we are considering, describe performance bottlenecks for CASPER, and give a high-level design for its parallelization. While many of the concepts here are common for parallel programming, we find this organization of options and challenges useful in specifying and understanding a design and try to reference features common to many planning and scheduling problems and algorithms. The paper adds preliminary experimental results to prior reported work (Clement & Estlin, 2010).

Planning Data and Functions

There are many kinds of planning and scheduling algorithms and some with specialized data structures. We will describe general and specific kinds of data and functions that a planning system may have and may want to parallelize, but we will do this from the perspective of CASPER, the planning system for which we have designed parallelization.

CASPER is a continual, local search planning and scheduling system. It is continual in that it re-plans continually based on state/resource updates from its environment. It is local search in that chooses its next search state (schedule) as a change from the current search state. Typically, CASPER loads either an existing schedule or abstract goal activities as the instantiation of a problem. Often, it initially uses a heuristic algorithm to refine any abstract activities into an initial, fully detailed schedule with state/resource values projected over a specified time horizon. This schedule is fully committed, in that it grounds values for start times, duration, resource usage, and other parameters. Then, CASPER iteratively updates the system state from the environment, projects future states, identifies flaws of the plan (constraint rule or preference violations/deficiencies), picks a flaw, and repairs or improves it using stochastic heuristics (Chien et al., 2000a, Rabideau et al., 1999). In this way, CASPER explores the search space of possible plans (good and bad) with only one schedule in memory. Alternatively, a planning system may systematically search for a plan, keep alternative search paths in memory, search in "action space" by building a plan action-by-action, postpone decisions on timing, ordering, or resource use, never introduce a new flaw, or backtrack when there are no valid choices.

From this point of view, Table 1 lists the kinds of data and associated functions that a planning and scheduling system may implement. The memory used by a planner is often dominated by search states. CASPER typically has a single search state, the current schedule, which takes up the majority of the process's memory. Other search states may be alternative parts or modifications to a plan, such as actions to include, their parameters, resource usage, temporal constraints, or "no-good" values (search sub-spaces to avoid).

Different planning systems may dynamically generate different types of data and may cache and delete different types. For example, in CASPER, flaw information is cached so that it is not all recomputed on every cycle, but valid intervals are re-computed each time an activity is scheduled or moved.

Parallelization Options

Because of the potential impact of data caching on overall performance, it is important to consider the parallelization of data as well as the parallelization of function. In addition to determining how to divide or duplicate data and function, a design must specify how the data is accessed (both locally and remotely) and how results are combined. Figure 1 summarizes the options for parallelizing planning and scheduling. The rest of this section will discuss each of these choices.

Parallelizing by Dividing Memory

As mentioned in the Introduction, accessing local cached memory is important for memory-intensive applications. What is memory-intensive depends on the hardware. For the Tilera TILE64™, each core has two 8KB L1 caches, one each for data and instructions, and a unified 64KB L2 cache (Ungar and Adams, 2009). For certain problem domains CASPER's schedule data can require hundreds of megabytes, so it will be difficult to minimize cache misses and keep them from dominating performance. In CASPER, we have seen cache misses make a 3X difference in performance for runs of the same problem.

For the Tilera processors, a planner can parallelize data across CPUs in the L2 caches. While conceivably any of the data in Table 1 could be parallelized by their attributes, activity, timeline, and temporal constraint information typically dominate memory allocation. Many distributed and multiagent approaches are based on memory division

distributing memory

- which data (search space, plans, activities, timelines)
- how to partition for load balance
- data replication

parallelizing computation

- which functions
  - entire algorithm
  - parts of algorithm
    - identifying valid search operators (valid intervals)
    - performing a planning/search operation
    - parameter dependency updates
    - state/timeline updates
    - identifying flaws
  - methods of data objects
  - data structure operations
- symmetry (loop-parallelized, master-slave, distributed)

data access and communication

- access location types (processing node, cache, RAM, disk, network/messages)
- allocation control (specify node, specify cache, OS decides)
- movement of data
- maintain consistency of replicated data (transactions/mutexes, conflict resolution)
- integration of results (transactions/mutexes, conflict resolution)
- data routing (centralized, hierarchical, peer-to-peer)
- synchronous or asynchronous
- communication services (hardware specific, threads, socket, file I/O, MPI, Linda, CORBA, database, distributed system interfaces)

Figure 1. Summary of options for parallelizing planning and scheduling.

or ownership. For example, an agent may only know about its own activities. In distributed constraint satisfaction and optimization, variables are allocated to agents, and, in the extreme, each variable is an agent.

Parallelizing by alternative plans/schedules

Depending on the planning algorithm and how it is parallelized, there may be multiple plans being explored at the same time. These may be alternative complete plans in a search space of plans (plan space), or they may be partially constructed plans in a search through action-space. In forward expansion action-space search, alternative plans may be the same up to a certain action or point in time. If memory is not shared, then the common parts of the plan may be duplicated across processing nodes. Instead of alternative plans for the same problem, there may be alternative sub-problems, such as with contingency planning. CASPER was used to generate command sequences for the Orbital Express mission that treated contingencies as completely separate problem instances, each corresponding to a combination of uncertain values for activity parameters (Knight, 2008). The section on Parallelizing computation discusses this parallelization of the algorithm at the highest level.

Parallelizing by time

One way alternative schedules may share common plan pieces is a division of a plan/schedule by time. For CASPER, the schedule could be divided by grounded time points, but a least commitment planner may choose to divide a partial order plan by minimum cuts of a directed acyclic graph formed by causal links or simple temporal constraints. Boundary handling is an issue for constraints and activities that span time divisions. This is one of many issues in how to share data, merge solutions, and access the entire search space, which we discuss later.

Parallelizing by resource/state timeline

Each processor may be in charge of a different subset of timelines, each timeline representing the projection of future values of a state/resource variable and its constraints (as discussed for CASPER in the section on Planning Data and Functions). This would be advantageous when a timeline's data is small enough to all fit in the local cache.

Parallelizing by activity

Activities and their associated data often consume the majority of memory for CASPER and other planning systems, so spreading activities across processors might be a way to avoid communication across nodes or, in the case of a multi-core processor, to find the data in the L3 cache and avoid accessing RAM. There are many ways to group activities, but does it matter? If the functions of the processing nodes are independent of any particular activity data, then grouping may not matter. If the nodes perform all operations involving their locally allocated activities, then it may be more efficient to group activities with those whom these operations affect. Grouping by time is an example of this. Activities may also be grouped according to temporal constraints or parameter dependencies, as they are often grouped in branches of a task network.

Activities may also be grouped by states and resources they commonly constrain or affect. Thus, grouping the timelines with the activities that constrain or affect them may be a good overall memory partitioning. If an activity affects timelines from two different divisions, then there is a choice where the activity is allocated. If the memory is not shared, copies of an activity may exist in multiple nodes. Options for sharing memory are discussed later.

Load balancing for data

The disadvantage of data parallelization is load balancing. While we just suggested that localizing activities and timelines that interact would minimize communication costs, at the same time, this strategy could produce computation bottlenecks. Some activities have no choices and will rarely need to communicate for updates. Likewise some timelines may capture exogenous effects such as day/night states and will never need to propagate. Groups of activities and timelines that require frequent attention and represent data bottlenecks may be best partitioned into smaller groups and (for multi-core processors) allocated in neighboring processing nodes.

Partitioning data for load balancing may be different for different scenarios and may even benefit from dynamic rebalancing. For example, time window divisions may iteratively shrink and expand depending processor load.

Parallelizing Computation

Where different functions can execute does not depend on where the data is stored if there is a communication path to access the data. But, the decision should not be made independently when the time to access the data depends on location and affects performance (as we discussed is the case for CASPER). Even still, it is not obvious how functionality might be distributed among processing nodes. In the section on Parallelizing by alternative plans/schedules, we began to discuss how sub-problems could be subdivided and solved separately. Possibly the simplest way to parallelize computation is to run the entire algorithm in parallel. A stochastic search algorithm (like CASPER) could copy the same problem to different nodes, each with a different random seed. Table 2 gives a design for this parallel stochastic search.

Figure 2 shows preliminary results for two of twenty different domains for which parallel instances of CASPER were run with different random seeds on 1, 2, 4, 8, 16, and 32 cores of the TILE64™. The filter model is an extreme case where running four instances could provide more than an order of magnitude speedup on average. The 3cs domain is another extreme where higher numbers of processors indicate poorer results because the performance appears to drop when more are running in parallel. We suspect this is a result of contention for memory since 3cs uses more memory than others. This drop in performance

	parallel stochastic search	parallelize by time
target platforms	network cluster and/or multi-core	mesh multi-core
memory distributed	schedule	activities, timelines by time
load balance strategy	none	iteratively resize window; eliminate nodes with no flaws
replicated data	problem	none
functions parallelized	entire algorithm	entire algorithm
symmetry	full	full
data location	RAM and disk	local cache, OS controlled
data movement	none	rescheduled activities, window resizing
replicated data	problem update	none -shared memory
integration	best wins and replaces all	constraints spanning time windows handled by eliminating nodes making no progress
data routing	copy in-memory or from file	peer-to-peer, adjacent nodes for adjacent time windows
synchronization	after fixed duration	asynchronous, exclusive access to data
services	SHAC, fork or Pthreads	SHAC (Clement and Barrett, 2003)
advantages	good for large search space on a cluster with a lot of memory	good for activities with time localized constraints/ preferences; may keep nearly all data in local cache
disadvantages	uses too much memory to run multiple problems on a single multi-core machine; redundant search for small search spaces	poor for non-localized constraints; local cache may not be large enough for both instructions and data, but that may be unavoidable

Table 2. Example stochastic and time-parallelized designs.

per core for greater numbers of cores is seen for many domains, but most, unlike 3cs, appear to expect some speedup for greater numbers of cores.

Figure 3 plots the minimum and maximum run times for the twenty domains on eight cores, above which performance does not typically improve. There are two memory controllers at the top and two at the bottom of the TILE64™ matrix of cores. The figure shows results for when the eight of the top cores and for when four are located at the top and the other four at the bottom. For example, the tireworld domain (plotted to the left in Figure 3) runs between 0.03 and 0.12 seconds when the eight cores are all located together at the top of the processor. When the cores are divided between the top and bottom, run times ranged from 0.01 to 0.03 seconds. The common effect of splitting between top and bottom is that the variance in run times is lower. Like tireworld there are often also improvements in run times, although typically small. While we have not determined the cause of these effects, a hypothesis is that crowding more processes around fewer controllers increases contention.

Another approach to parallelizing computation is to solve the same problem on each node but with each exploring non-overlapping solution spaces. For example, different nodes could explore different assignments of resource alternatives to activities, disallowing them from switching resources. Planning algorithms that employ dynamic programming or Markov models can often be applied recursively to connected subgroups of states in both expanding the state-action space and evaluating them. Recent work parallelizes the planning search space by iteratively farming out search states for expansion (Burns et al., 2009, Kishimoto et al., 2009).

There are many more possibilities when considering how to distribute and replicate parts of the algorithm (such as those listed in Figure 1). A simple concept for taking advantage of data parallelization is to restrict object methods to be invoked on the processor to which the data is allocated. Another potentially easy way to get some parallelization speedup is to parallelize basic data structure functions such as `sort()` or `find()`. An experimental Parallel Mode extension to the GNU C++ Standard Library supports parallelization of functions such as these.

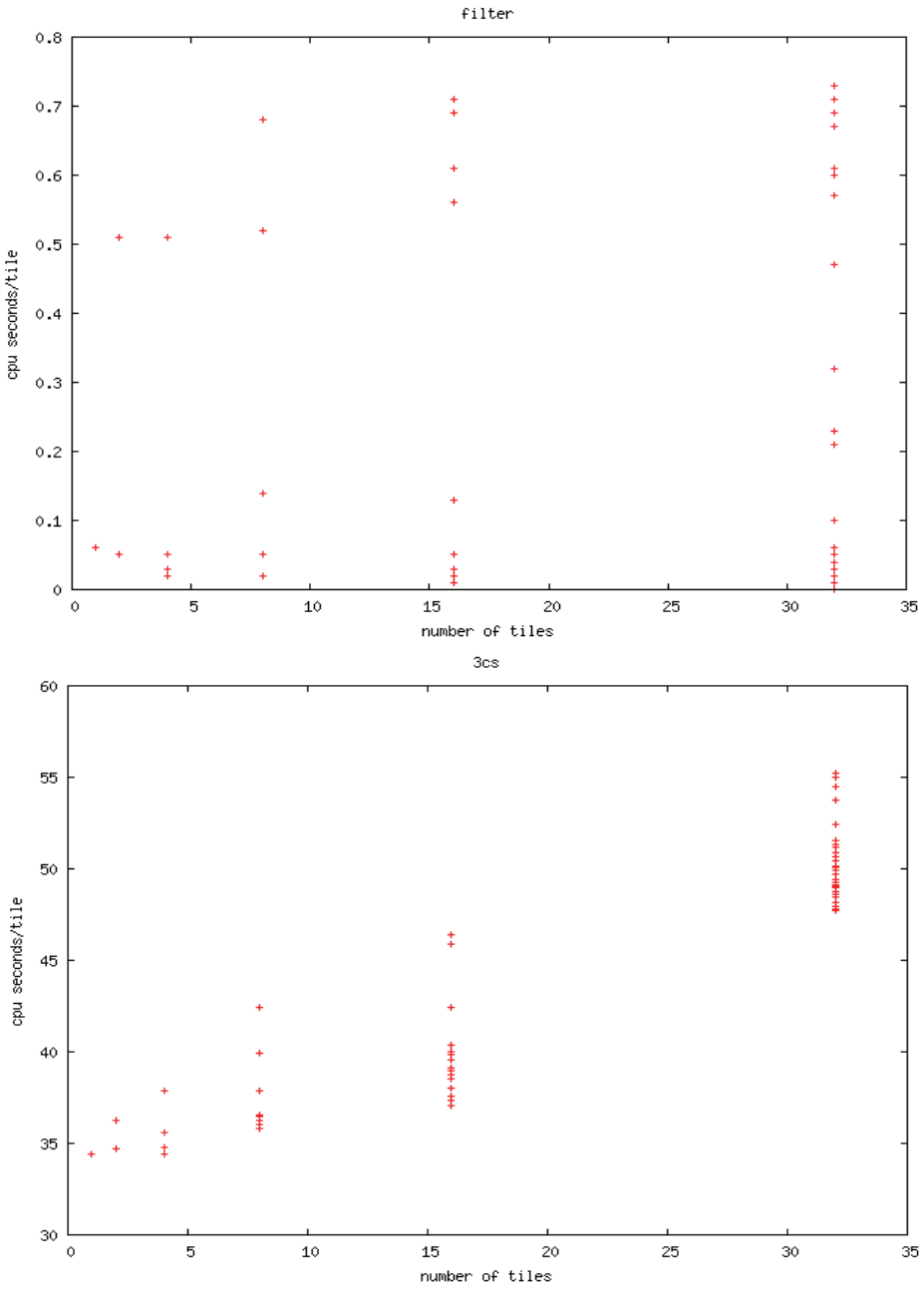

Figure 2. Run times for parallel CASPER instances with different random seeds on 1, 2, 4, 8, 16, and 32 TILE64™ cores for the filter (left) and the 3cs (right) domains. The x-axis is the number of cores. The y-axis is run time in seconds.

Symmetry

The functions listed in Table 1 could be parallelized symmetrically and/or asymmetrically. One kind of symmetric parallelization would be running the entire algorithm for different sub-problems. An asymmetric master-slave parallelization may have a centralized planner farming out search operations to other processors. A completely asymmetric distributed planner could dedicate processors to each one of the functions in Table 1.

Communication for Parallelization

Choices of how to assign data and function are often not clear until it is decided how the processing nodes communicate. For example, running the planning algorithm in each of a series of time windows can be done in many ways. Activity and timeline data could be divided into the caches of the corresponding processor, and activities overlapping time windows could be locked so

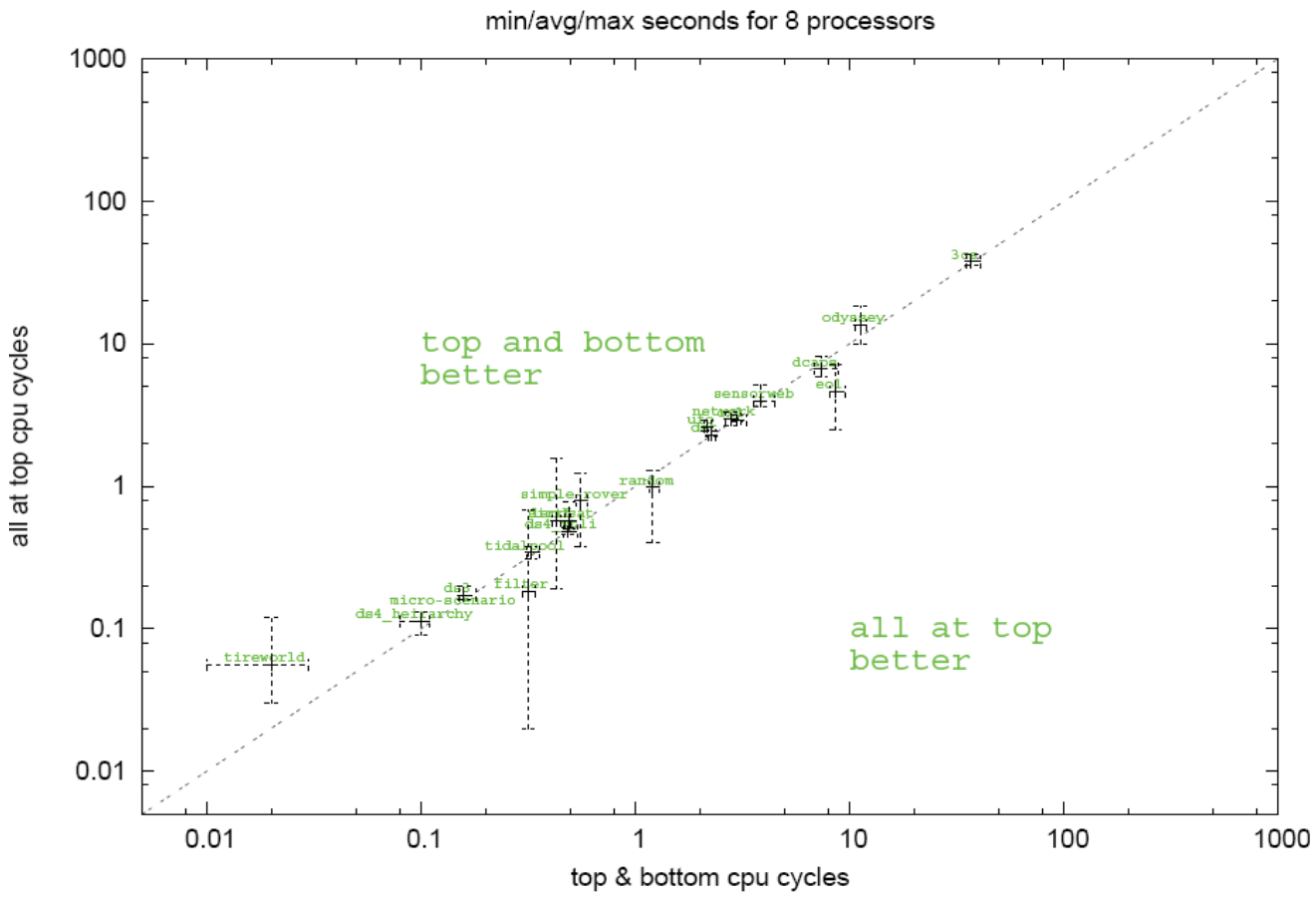

Figure 3. Minimum and maximum CASPER run times on eight TILE64™ cores for twenty scheduling domains and for two core choice schemes: one with all eight cores together and one with the cores are split into two sets of four on opposite sides.

that communication between cores is only necessary to retrieve timeline updates in shared memory. Alternatively, processing nodes could be launched as separate processes, let the OS decide where to allocate the data, allow activities to move into other windows, and reference all data and containers through objects that either contain the data or retrieve it over a socket interface.

Replicating data and maintaining consistency

These communication choices literally tie the other design choices together. One fundamental choice for parallelizing any subset of data and/or functions is whether the data is replicated in different locations or accessed just from a central location. A location can be from a processing node or shared memory. Shared memory could be located in cache, RAM, or non-volatile storage (e.g. a hard drive). The location could be at a remote processing node or this one.

If data is replicated, then there must be a strategy for keeping the data consistent across nodes. For example, if flaws are being fixed in parallel, one node could be rescheduling an activity based on information that another has since changed. One way to handle this is to treat the entire computation and subsequent fix as a single transaction. Mutexes could be used to lock other nodes out from the particular activities and timelines being read and written. However, efficiency can suffer from long wait durations, and care must be taken to avoid deadlock or starvation. Another approach is to allow one node to reschedule the activity based on possibly bad information. At some point later, all nodes must come to consensus (or de-conflict) to ensure a valid solution. Similarly, when the results from different nodes overlap in information, merging may require de-confliction of disagreeing results.

Routing data

A parallel planning design may also need to consider how data is routed among the nodes. The design may include customized routing (such as a message protocol that passes tokens around a ring of nodes). The design may not include routing and delegate it to the network, operating system, or hardware, depending on the physical layout of the nodes.

The design can also passively specify routing by selecting which processing node gets which data and functions. For example, if timelines are divided by time

on a mesh multi-core processor where memory access time is proportional to the distance the data must travel, then adjacent timeframes could be mapped to adjacent processors so that changes propagate as quickly as possible. This is a peer-to-peer or, more specifically, a chain flow of information.

Information may also flow to and from a central node. For example, as a stochastic local search planner, CASPER could run the same problem on different processors with different random seeds. A single node may be receiving state updates and could be in charge of sending out the current state and schedule to the other nodes, collecting scores, loading the best solution, and repeating by sending out the new state and schedule.

The flow may be more efficient in a hierarchy. The Map-Reduce concept includes parallelizing the transmission and collection of information. So, instead of a single node sequentially talking to each other node, the node could spread and merge information through a tree of nodes, reducing the overall communication time from linear to log of the number of processors. Planners that model problems as Markov decision processes could assign sub-graphs to processing nodes to naturally parallelize the Bellman backup algorithm in merging results.

Synchronous vs. asynchronous communication

The merging of results may require some level of synchronous processing. For example, the Bellman backup requires that each node receive all downstream results before sending its result upstream. An algorithm that parallelizes by time may allow asynchronous updates between nodes. In general, nodes are consumers and producers of data, and consistency can depend on processing data from consumers after they produce it. The choice of synchronous and asynchronous communication is similar to that of replicating data. Waiting for updated data may avoid wasted computation of bad information but could result in wasted time computing nothing. The ideal design minimizes the dependencies between processing nodes to avoid both ways of wasting processing time.

Software for parallelizing C++

When it comes down to actually implementing a design, there are more options to consider. While there are programming languages especially suited for parallelization (such as Chapel, Fortress, X10, and functional programming languages like Haskell), we restrict our discussion of tools to those we have considered for parallelizing CASPER, which is written in C++ and supported for a wide variety of platforms. The general takeaway is that deciding what to use can be complicated.

For creating and synchronizing threads, simple choices include using fork() and wait(). POSIX Pthreads give a larger set of functions for managing threads, including mutexes and other synchronization functions. OpenMP takes another step in sophistication by providing functions for managing shared memory and directives to launch and manage groups of threads and their recombination. Experimental support of some of OpenMP's functions have recently become available in GNU C++Standard Library extensions.

For communicating across processes, possibly on different computers, another even larger set of options is available. Messages can be passed reliably over the Internet with socket libraries. Pipes can similarly be used for systems sharing a file system. A number of other message passing libraries using different protocols and syntax are available. CORBA allows objects to be shared over the network. There are also many software packages implementing communication among threads and processes using message passing and/or shared memory for a wide range of platforms. Some of these include MPI, Linda, and PVM. Multi-agent software architectures provide protocols that are more tailored to reasoning applications, like distributed planning. Shared Activity Coordination (SHAC, Clement and Barrett, 2003) is one such distributed planning framework that is integrated with CASPER. SHAC allows planning and scheduling to be coordinated using scheduling permissions and roles that can be manipulated to subdivide problems and ensure consistency.

Design for Parallelizing CASPER on a Mesh Multi-Core Processor

As shown in Table 1, typical bottlenecks of CASPER are related to computing valid intervals for choosing where to reschedule an activity, updating parameters in a network of functional dependencies (such as energy = power * duration), collecting parameters needing updates from dependencies, and, in select cases, detailing and re-detailing activity hierarchies. In Table 2, we list options for parallelization for two example designs we have discussed. Table 3 summarizes our current design as a combination of others for parallelizing CASPER for Tilera's TILE64™ mesh multi-core processor. This design is largely driven by the hardware, the bottleneck functions, and simplicity of implementation.

While we will likely run multiple CASPERs in parallel for evaluation since it is relatively easy to implement, we do not expect it to be a good design choice for a mesh processor because of CASPER'S large memory requirements. It would still be possible to spread memory across the processors by subdividing the problem (for example, by time as shown in Table 2). Also, instruction memory could either be replicated for local caches or remotely accessed from other cores' caches. However, with any of these options, requiring each core to execute the entire program can result in slower memory access compared to allocating smaller scopes of functionality to the cores. Moreover, the ability to subdivide a problem and balance processing can vary from problem to problem.

	parallelize bottleneck functions		parallelize repair/optimize by flaw type
memory distributed	**timelines**	**dependencies/activities**	none
load balance strategy	dynamic grouping		none needed
replicated data	none		
functions parallelized	propagation, valid intervals	propagation, flaw gathering	repair, optimize
symmetry	peer-to-peer		master-slave, asymmetric by flaw type
data location	local cache, pre-specified		RAM/cache
data movement	none		OS controlled
replicated data	none -shared memory		
integration	shared memory, no flaws		determine independence
data routing	centralized through cache & RAM		
synchronization	synchronize after propagation	full propagate before flaw gathering	sequential processing of dependent flaws
services	Pthreads		
advantages	may keep nearly all data in local cache		many flaws may be independently addressed
disadvantages	local cache may not be large enough for both instructions and data, but that may be unavoidable		difficult to take advantage of locally cached data.
	difficult to load balance and maximize utilization		

Table 3. Combined design for CASPER on mesh multi-core processor. Columns represent what to parallelize. Rows describe how to parallelize for different options.

Thus, it makes sense to try and distribute functionality and memory at a lower level. The next highest level of functionality that could be naturally distributed and uncoupled is the iterative repair cycle. This is our design choice in the right column of Table 3. Thus, two schedule operations could be performed in parallel. The danger is in maintaining consistency since flaws involving the same timelines and activities can be chosen contiguously. However flaws of different types often can be fixed independently without significant consistency issues. Thus, we plan to only parallelize flaws that we know do not interact or can force to not interact. For example, the rescheduling of two activities can be isolated to separate timeframes or choice of resource.

Discovering what combinations of operations are legal is a significant research effort in itself, so we plan to localize functionality according to the type of scheduling operation determined just by the type of flaw. For example, a processor may be allocated to only fixing flaws that require an activity be moved. Another may only fix stale dependency functions. Since each performs a limited function, the instruction cache is less likely to miss, but there is no guarantee that it will revisit the same data.

In order to address our bottleneck functions, we plan to additionally localize timeline and parameter dependency data and functions to separate processors as slaves to the overall CASPER application (shown in the middle two columns of Table 3). We believe that the data may be independently subdivided well enough to take greater advantage of the local caches. Since dependencies comprise the majority of the memory for activities, it makes sense to distribute the activities with their dependencies. Activities sharing dependencies will be grouped on the same core or a nearby core. Valid timeline interval, dependency propagation, and stale dependency collection will be parallelized in a simple MapReduce fashion (Dean and Ghemawat, 2004) by recursively spawning the function calls as threads across cores in a tree so that the combination of results is parallelized.

In order to localize both functionality and data, we must either keep track of which data is on which core to know where to spawn threads or set up persistent services on the cores, possibly a thread for each activity and timeline. Either choice complicates our design since we need to specify in our code to locate data and functions. We plan to spawn threads since having services would require the additional complication of a communication protocol.

Another challenge of this design is deciding how to map data and functions onto the processors and deciding whether to migrate data for load balancing. The TILE64™ has different caching schemes for three memory types: read-only, read-write, and user-managed. In short, read-write is the default and the data is assigned a specific core with no replication to other cores. User-managed allows

replication, but a write by one core invalidates the cache lines of the other cores' copies, and the data is written to main memory. We want to experiment with these schemes, but read-write requires that we determine a fixed allocation of data to cores up front. In order to migrate data, we would need reallocate its memory, or create copies in CASPER. Reallocation is possible when re-detailing activities to different decompositions, but we plan to avoid this complication.

Conclusion

We have described a large number of choices that may go into a design for a parallelized planning system. We describe our design for parallelizing CASPER, a stochastic, continual iterative repair planning and scheduling system for a mesh multi-core processor. In particular we characterize our design choices by listing what implementation features each would entail. We found it useful to characterize the design options to help us create a detailed design plan and to discover potential problems.

Acknowledgments

The research described in this paper was performed by the Jet Propulsion Laboratory, California Institute of Technology. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government, or the Jet Propulsion Laboratory, California Institute of Technology.

References

Ronen I. Brafman, Carmel Domshlak. From One to Many: Planning for Loosely Coupled Multi-Agent Systems. In *Proc. ICAPS*, 2008.

Ethan Burns, Seth Lemons, Wheeler Ruml and Rong Zhou. Suboptimal and Anytime Heuristic Search on Multi-core Machines. In *Proc. ICAPS*, 2009.

Steve Chien, Gregg Rabideau, Russell Knight, Robert Sherwood, Barbara Engelhardt, Darren Mutz, Tara Estlin, Benjamin Smith, Forest Fisher, Anthony Barrett, George Stebbins, and Daniel Tran, ASPEN - Automating Space Mission Operations, In *Proceeding of SpaceOps*. Toulouse, France, 2000a.

Steve Chien, Russell Knight, Andre Stechert, Rob Sherwood, and Gregg Rabideau. Using Iterative Repair to Improve the Responsiveness of Planning and Scheduling. In *Proc. AI Conference on Planning and Scheduling* (AIPS). pages 300—307, 2000b.

Bradley J. Clement, Anthony C. Barrett. Continual Coordination through Shared Activities. In *Proceedings of the Second International Conference on Autonomous Agents and Multi-Agent Systems* (AAMAS), 2003.

Bradley J. Clement and Tara A. Estlin. Exploring Parallelization Options for Planning and Scheduling. In *Proceedings of the 4th Scheduling and Planning Applications workshop* (SPARK) at ICAPS10, 2010.

Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation* (OSDI), 2004.

E. Ephrati and J. Rosenschein. Divide and conquer in multi-agent planning. In *Proceedings of the National Conference on Artificial Intelligence* (AAAI), pages 375–380, July 1994.

Michael P. Georgeff. Communication and interaction in multiagent planning. In *Proceedings of the National Conference on Artificial Intelligence* (AAAI), pages 125–129, 1983.

Akihiro Kishimoto, Alex Fukunaga, Adi Botea. Scalable, Parallel Best-First Search for Optimal Sequential Planning. In *Proc. ICAPS*, 2009.

Russell Knight, Automated Planning and Scheduling for Orbital Express. In *Proceedings of the 9th International Symposium on Artificial Intelligence, Robotics and Automation in Space*, 2008.

Peter Kogge, Tara Estlin, and Benjamin Bornstein, Energy Usage in an Embedded Space Vision Application on a Tiled Architecture. *Proceedings of the 2011 Infotech@Aerospace AIAA Conference,* St. Louis, MO, March 2011.

Amy L. Lansky. Localized Search for Multiagent Planning. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence* (IJCAI), 1991.

R. T. Maheswaran, M. Tambe, E. Bowring, J. P. Pearce, P. Varakantham. Taking DCOP to the Real World : Efficient Complete Solutions for Distributed Event Scheduling. In *Proc. AAMAS*, 2004.

G. Rabideau, R. Knight, S. Chien, A. Fukunaga, and A. Govindjee. Iterative Repair Planning for Spacecraft Operations in the ASPEN System. in *Proc. International Symposium on Artificial Intelligence Robotics and Automation in Space* (i-SAIRAS), Noordwijk, The Netherlands, June, 1999.

K. P. Sycara, S. Roth, N. Sadeh, and M. S. Fox. Distributed constrained heuristic search. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(6), pages 1446—1461, 1991.

David Wentzlaff, Patrick Griffin, Henry Hoffman, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. On-Chip Interconnection Architecture of the Tile Processor. *IEEE Micro*, 27(5), pages 15—31, 2007.

Michael Wolverton and Marie desJardins. Controlling Communication in Distributed Planning Using Irrelevance Reasoning. In *Proceedings of the National Conference on Artificial Intelligence* (AAAI). pages 868—874, 1998.

David Ungar and Sam S. Adams, Hosting an Object Heap on Manycore Hardware: An Exploration. In *Proc. Dynamic Languages Symposium* (DLS), pages 99—110, 2009.

R. Zhou, and E. Hansen. Parallel Structured Duplicate Detection. In *Proceedings of AAAI-07*, 1217–1223, 2007.