Globally Parallel, Locally Sequential*

A Preliminary Proposal for Acumen Objects

Paul Brauner
Rice University
paul.brauner@inria.fr

Walid Mohamed Taha
Halmstad University
walid.taha@hh.se

ABSTRACT

An important and resource-intensive class of computation codes consists of simulators for physical systems. Today, most simulation codes are written in general-purpose imperative languages such as C or FORTRAN. Unfortunately, such languages encourage the programmer to focus her attention on details of how the computation is performed, rather than on the system being modeled.

This paper reports the design and implementation of a novel notion of an object for a physical modeling language called Acumen. A key idea underlying the language's design is encouraging a programming style that enables a "globally parallel, locally imperative" view of the world. The language is also being designed to preserve deterministic execution even when the underlying computation is performed on a highly parallel platform. Our main observation with the initial study is that extensive and continual experimental evaluation is crucial for keeping the language design process informed about bottlenecks for parallel execution.

1. INTRODUCTION

Hardware manufacturers were quick to warn us that the arriving multi-core systems will be harder to program. Despite this early warning, there is still no emerging technology for programming such multi-core systems. Many researchers have advocated the idea of designing new programming languages to harvest the power of multi-core systems [2, 8, 3], but there is no consensus on a single most effective approach.

Our interest is in finding an abstraction mechanism that is suitable for modeling physical systems in such a way that they are easy to execute in parallel. This exploration is carried out in the context of Acumen, a physical modeling language that exhibits both continuous and discrete behavior [9, 1, 7]. While Acumen was not originally conceived of as a parallel language, as soon as the first major release of the language was completed, it became clear that simulations of physical systems quickly become computationally intensive. This problem gave rise to the need to understand the extent to which various language design choices can enhance or erode the ease with which programs written in this language can be executed in parallel.

*This work was funded by National Science Foundation research grant #0720857 entitled "Building Physically Safe Embedded Systems".

1.1 Globally Parallel, Locally Sequential

Physical systems themselves are naturally highly parallel. One can go further and speculate on the properties of the physical world that enable this natural ease with which the physical world supports such concurrency. For example, it may be that there is a tendency for physical distance to diminish interactions between physical objects.

To what extent can this informal observation guide us to a language design that promotes a programming style that is simultaneously intuitive and easy to execute on parallel platforms?

This paper summarizes the initial results of this exploration. The focus is on the design of a notion of an object that would encourage a "globally parallel, locally sequential" style of computing. Our hope is that this style can be simultaneously intuitive for modeling the physical world and at the same time well matched to the way parallel computing systems are built.

1.2 Contributions

To begin addressing the second part of this question, that is, matching parallel computing systems, we design and implement a language featuring a notion of objects aimed at promoting the "globally parallel, locally sequential" style (Section 2). We evaluate the performance of a simple strategy for the parallel execution of this language (Section 3). The preliminary results suggest that the approach bears promise but that there are at least two technical challenges that must be overcome in order to demonstrate the viability of this idea.

2. ACUMEN OBJECTS

Acumen is a modeling language for describing hybrid (continuous/discrete) systems. Initial work on Acumen focused on supporting undirected equations for modeling continuous behaviors [9]. From this early work, it became clear that the language can benefit from

- An abstraction mechanism for modeling physical objects
- Parallel execution

- Closer coupling between continuous and discrete behaviors.

A natural question is whether all three goals should be combined into one abstraction mechanism. There are in fact several reasons not to use an abstraction mechanism to address multiple concerns. For example, one often gets a simpler formal semantics if different features of a language are made orthogonal. One can also argue that it should be the job of the compiler to devise a strategy for parallel execution given any model that the user writes.

Our view is that there is a need for abstraction mechanisms that bridge the gap between the real-world systems that the user is modeling and the computing machines that are available for simulating these models. Our approach can be viewed as searching for an exoteric abstraction mechanism that can be widely understood by a large number of users and that at the same time carries varying levels of esoteric semantics that are relevant to increasingly smaller groups of increasingly specialized users.

To emphasize that the exoteric aspect of the design is the more significant component of the work, and that physical reality should be a key source of inspiration for this design, we will describe such an abstraction as "natural objects." In the rest of this section, we propose a design and an implementation strategy for such a notion.

2.1 A Design for Natural Objects

Our current proposal is for natural objects that have the following properties:

1. Identity. Behavior does not determine equality; in particular, two objects with identical behavior can still be different.

2. Dynamic existence. Objects can be dynamically created and deleted.

3. Hybrid state. The values of variables can change both continuously and discretely.

4. Composition. Each object has a notion of children. When an object is created, it is a child of the object that created it. A mechanism is available for moving objects from one parent to another.

5. Internal laws governing behavior. While it is possible for objects to interact with other objects, their primary character is determined by a set of internal laws. A mechanism is provided to allow the user to specify the behavior of the object in a single "method" that can be viewed as a "mode handler." Intuitively, the mode handler executes in every instance as continuous time advances.

6. Locally sequential behavior. The execution of a mode handler is sequential and can be used to modify the state of the object or the state of its children. An object can modify only its own state, or that of its immediate children.

These features enable highly concise descriptions of complex systems. For example, through dynamic objects we can easily model the creation and evolution of highly diverse populations of objects. Through hybrid states we can model multi-modal systems that can be as simple as a ball that has different dynamics when airborne and when bouncing, as complex as an autopilot system that uses radically different models for sub-sonic and supersonic flight. Composite objects enable the hierarchical description of assemblies as well as the rules that govern the interaction between objects that are children of the same parent object. Limiting communication of any object with its children gives concrete meaning to the notion of composition as an encapsulation mechanism, and it makes the parent's mode handler the right place to specify the physical laws that govern the dynamics of the objects at the highest level. In this model, when the laws can be decomposed to small principles that only involve sub-assemblies, they can be pushed down to sub-assembly objects, and this makes it explicit that these rules only constrain a specific subset of the grand children of a given object. From the implementation point of view, this restriction on communication is highly significant, as it encourages the author of Acumen models to ensure that there is significant locality of reference in the communication taking place in the overall model. Finally, having one mode handler "method" means that there is one clear place where the user specifies what governs the evolution of each object.

2.1.1 Hybrid Behavior

Hybrid behavior combining both continuous and discrete behaviors arises naturally in models of simple systems such as that of a bouncing ball that reverses direction as soon as it hits the ground. Hybrid behavior raises several questions, including how to resolve situations in which there is both a continuous rule and a discrete rule assigning a value to a variable. In the current semantics for Acumen, the issue is resolved by treating all such rules as ordered in the same order that they appear in the source program. This choice restricts the semantics of such rules to be sequential, and requires that fully sequential execution be viewed as the correct semantics. One can argue that a better semantics would exclude rules where such order matters. This is an attractive possibility, since it could both make reasoning about such programs easier and make them easier to execute in parallel. However, such improvements can be retrofited on top of the current semantics without compromising determinacy of the semantics, even in the parallel setting.

2.1.2 Deterministic Parallelism

In a sense, defining a language that should only have a deterministic parallel semantics is easy: one just needs to define a single, sequential semantics and make sure that all of its components are deterministic. In practice, however, the real challenge is to make sure that such a semantics does not introduce unnecessary constraints that would *force* any execution to be sequential. This is where much of the effort went in designing this semantics for Acumen.

The most basic source of non-determinism in concurrent systems is shared mutable state. In Acumen, all interactions between objects happen by changing the state of a variable visible to both objects. Non-determinism is avoided by allowing only the object and its parent to see an object's state, and requiring that the parent and the child never run in parallel. "Global parallelism" is enabled by allowing all children of any object to run in parallel. The user specifies all communication between such objects in the mode handler of the parent object.

Shared mutable state is not the only challenge for determinacy. For example, the semantics of dynamic object allocation must be defined so that it can, in principle, be performed by completely independent threads concurrently, and without having to lock a centralized store. Similarly, simulations often involve logging of various variables over time, and the semantics is defined so that logging can be done locally in the run time representation of various objects without having to pass around the data globally.

To help facilitate a possible future parallel execution of mode handlers themselves, the language is designed to have a clear separation between expressions and statements, and all expressions are free of imperative side-effects.

We expect that the language will eventually be statically typed. For the time being, however, the language is fully dynamically typed. Typing is orthogonal to the issues discussed in this paper.

2.1.3 Dangling Pointers

Pointers arise as a result of having a notion of objects, and the fact that the user is able to explicitly create and terminate objects at specific instances in time. An interesting technical problem that arises in the implementation of the language is avoiding dangling pointers. The problem is avoided by ensuring that objects can only terminate other objects that happen to be their children, and that termination is not allowed if this requirement does not hold.

2.2 Implementation

Two implementations of Acumen were built. To make the results of our research easily available to others, the implementations will be released under both GPL and BSD licenses. To make the implementations easy to install, they are implemented using the JVM platform, and they are written in the Scala language, which compiles to JVM bytecode.

2.2.1 Sequential, Reference Implementation

The reference implementation is written in Scala as a purely functional monadic interpreter. It constitutes a direct codification of an operational semantics for Acumen. This implementation was created primarily to facilitate unit testing of the parallel implementation, which is necessary to help ensure that the implementation is consistent with our intended high-level semantics. Purely functional data structures are used to explicitly model the heap where Acumen's objects are represented during the simulation. However, because of the use of such purely functional data structures, and because we did not invest the time in tuning the performance of these data structures, this implementation is not well suited even for sequential execution of Acumen programs. The purely functional data structures also make it difficult to imagine how it can be useful for parallel execution without potentially significant duplication and communication overhead. Thus, an interesting byproduct of writing this semantics is that it shows that writing a program in a purely functional style does *not* automatically mean that we are able to avoid all unnecessary sequencing constraints.

2.2.2 Parallel Implementation

To explore the feasibility of automatic parallel execution of Acumen programs, a second, independent interpreter was implemented. This implementation uses Scala objects to represent Acumen objects, and Scala assignments are used to implement all state updates in Acumen. While this interpreter implementation can be expected to be significantly slower than a compiled implementation, the two implementations can be expected to scale similarly as we increase the level of parallel threads available. As such, the interpreted implementation seems suitable for preliminary evaluation of the scalability of the language.

2.2.3 Scheduling Strategy

A simple strategy is used for scheduling. The user indicates the number of threads (cores) available; then the execution of an Acumen program consists of starting at the root object, executing its mode handler, and then distributing the available threads among all of the root object's children. This is applied recursively.

3. EXPERIMENTAL EVALUATION

We focus this preliminary evaluation on artificial examples that quickly generate a large number of dynamic objects. Three benchmarks are used, all of them modeling a ball that repeatedly breaks into two smaller parts each time it bounces off a surface. The benchmarks are minor variants of each other in 1, 2, and 3 dimensions (full source code is available online [6]). The repeated breaking behavior means that as time goes by we get an exponentially larger collection of objects.

3.1 Experiments

All experiments were carried out on a Mac OS X Xserve3,1 server with 2 Quad-Core Intel Xeon of 2.93 GHz with four processors each (8 cores in total) and 24 GB of RAM. The processor Interconnect Speed is 6.4 GT/s. The operating system is version 10.6.4, build 10F569. Given a fixed number of threads, each example is run once as a warm-up phase, and timings are then based on 10 successive executions. The experiment is run with threads varying from 1 to 8.

3.2 Results

The total run times (normalized with respect to the 1-thread case) are as shown in Figure 1. Timings for the 1-D benchmark show modest speed-ups as we go from 1 to 2 threads, and the trend continues until we reach 8 threads, when there is a slow-down. Timings for the 2-D benchmark are similar, although both types of change are even more muted. Timings for the 3-D benchmark show a more significant speed-up when going from 1 to 2 threads, followed be negligible improvement when going to 3 threads, and then from 4 to 8 threads the total performance deteriorates at an almost linear rate.

4. ANALYSIS

The limited speed-up and (in some cases) slow-down was clearly unexpected. However, these experiments were important in drawing our attention to a number of features of the design of Acumen that deserve closer scrutiny: parallel aggregate structure and load balancing.

4.1 Parallel Aggregate Structure

The three benchmarks are an easy way to push the implementation in terms of creating a large number of objects. The idea was that an abundance of objects would make it easy to demonstrate the effectiveness of our simple scheduling strategy, and that more careful scheduling

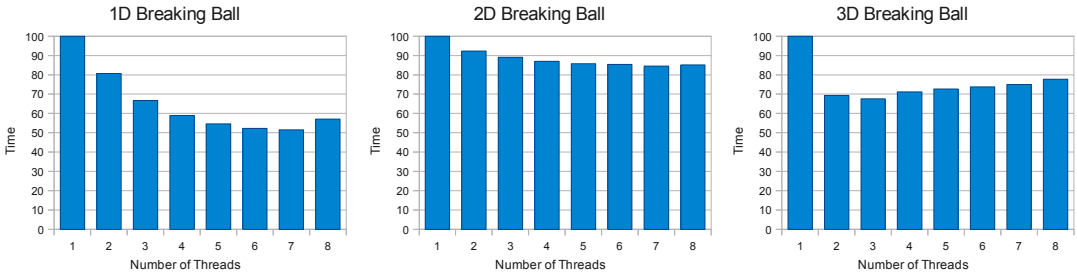

Figure 1: Experimental Evaluation

would be needed only when the number of objects is relatively small. Profiling drew our attention to the current implementation of allocation as well as the updating of the parent object "children" list. In fact, the current implementation for the children list is sequential, as illustrated by the following fragment of the implementation:

```
prt match {
  case Some(resp) =>
    resp synchronized { // sync here!
      resp.children = resp.children :+ res
    }
  case None => ()
}
```

Thus, the children list is a likely bottleneck for all the benchmarks. In our current semantics for Acumen, the rule is that the grandparent "inherits" the grandchildren if the parent dies. This means that for the all breaking ball examples, the root object has a children list that doubles in size after each generation.

To validate this hypothesis, we modified the 2-D example by stopping the breaking behavior once "ball mass" reaches a certain point. As the data presented in the first of the three graphs of Figure 2 shows, this change leads to a dramatic change in the behavior of the benchmark and lends credibility to the idea that a large number of objects in the simulation enables exploiting of the available resources. But it also clarifies our understanding of this hypothesis, in that it currently holds primarily when this large number of objects is constant. The observation seems to be further confirmed by the performance of another diagnostic benchmark that we called Flat Tree [6], which creates 100 objects at the start of the simulation and keeps that number constant until the end of the simulation. As the second graph of Figure 2 shows, significantly better scalability is attained in this kind of situation.

4.2 Load Balancing

The benchmarks also suggest that more sophisticated load balancing may be needed when the number of objects in the simulation is "small." They also draw attention to the fact that the number of objects can be "small" in two distinct ways, even though their absolute number is quite large. The first case is when the number of objects is relatively small with respect to the number of available threads/cores. The second is when the object tree has relatively small fan-out as we go down the tree, in which case the number of objects can be considered to be structurally small.

Relatively Small Number of Objects.

We will say that a set of objects have a relatively small number (or "tasks") when their number is small compared with the number of available resources (such as threads or CPUs). Poor utilization of the latter can result from the basic strategy of focusing on parallel execution only for completely independent tasks that consist of the execution of the mode handlers of different natural objects. The performance numbers hint at this problem through the sudden, abrupt steps that we see as we increase the number of threads. The worst case of this problem arises when we have only one task and an arbitrarily large number of processing units. In this case, total utilization goes down to almost 0. In particular, it is the limit of $1/t$ as t grows indefinitely, where t is the number of threads.

An unfortunate feature of this problem is that the number of tasks can still be "small" even when it grows to a large size. For example, and assuming we have n threads and t equal-sized tasks, even when n of tasks is just slightly over t — let's say, $t+1$ — the utilization falls down to $(t-1)/2 \times t$, which is 50% at the limit. In general, if we have n that is strictly between $i * t$ and $(i+1) * t$, then we have $(t-1)$ unused time steps, which means that for large numbers we can use the limit of $(t-1)/(i+1) \times t$ which converges to $1/i$, essentially the ratio t/n between the number of tasks and the number of threads. Depending on the number of tasks in the problem, even for 1000 threads this can be a large fraction and can be viewed as a direct penalty for using only a basic scheduling strategy.

Structurally Small Objects.

A more subtle variant of the above problem arises when the relative number of tasks is not small, but the object tree is structured in a way that prevents even distribution of objects between threads. In particular, in our benchmarks, when a ball breaks, it is replaced by two balls that become immediate children of the main simulation task. However, we expect that there will be many simulation problems in which there is an abundance of composite objects that consist of smaller objects. Because our strategy is to recursively distribute threads as we go down the object tree, in essence the relatively small object number problem can occur at any level in this process of recursive descent over the tree. To confirm this point, we constructed a diagnostic benchmark that creates a tree of objects called Deep Tree [6]. This example recursively generates a tree with a total number of 256 nodes, but with fan-out of exactly two at any level. As can be seen in the third graph presented in Figure 2, the problem of poor overall utilization shows up in this example

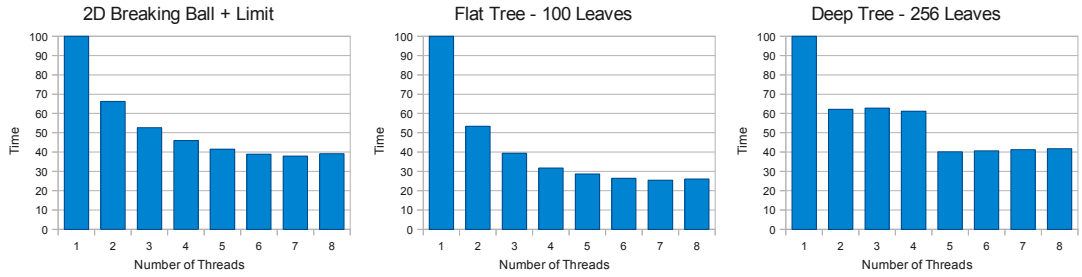

Figure 2: Diagnostic Benchmarks

as well. Thus, any solution to this problem must also apply to its recursive variant. In fact, from the point of view of scalability, it is likely to be much more important that the solution works reasonably well for the recursive variant of the problem than that it works particularly well for the flat variant. We expect that a solution to this problem could be Cilk-style work stealing [5, 4].

5. CONCLUSION AND FUTURE WORK

We have described ongoing work that explores the idea that a language can be designed to encourage writing simulation codes in a way that makes them easier to execute on parallel architectures. A prototype implementation of a language based on this idea was built, and preliminary experimental measurements were gathered. The results are timely, in that they draw our attention to aspects of the design that we had not previously expected to have a drastic influence on parallelization.

The first issue relates to allocation-intensive computations. Addressing this problem may require using decentralized and distributed memory primitives, a parallel structure for representing the children list, or a combination of both.

The second issue is that a recursive strategy for the parallel execution of Acumen when the task set is "small" can be problematic. Here, "small" means that the task set is a small multiple of the number of threads available.

While these were direct lessons learned from these experiments, there were also other, somewhat less immediate ones. For example, the experiments draw attention to the fact that one task can keep all threads/CPUs waiting. This means that there will be situations in which the only way to speed up a computation is to parallelize within a task. Once we have achieved reasonable success in terms of load balancing at the task level, it may be necessary to reconsider the issue of parallel execution inside tasks.

Especially if the process of scheduling resources becomes complicated, it will be interesting to see if we can exploit the fact that task sizes and dependencies will often vary significantly between iterations of a simulation, and can therefore be scheduled more efficiently and more accurately than what we would get if we treat them all in the same way. As was noted in the discussion on structurally small objects, it is particularly important that this solution work effectively for the recursive variant of this problem.

Acknowledgments. We are very thankful for many excellent and helpful suggestions on this work from John Mellor Crummey, Vivek Sarkar, Kei Davis, and the participants of the POOSC/SPLASH 2010 Workshop. A special thanks goes to Julien Bruneau and Alexandre Chapoutot for being the first two external users of this new implementation of Acumen. Their programs and their patience helped us greatly in testing the semantics. Jerker Bengtsson and Alexandre Chapoutot kindly read and commented on a draft of this paper.

6. REFERENCES

[1] Acumen page. `http://www.acumen-language.org`.

[2] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. The Fortress Language Specification. Technical report, Sun Microsystems, Inc., 2007.

[3] J. S. Auerbach, D. F. Bacon, P. Cheng, and R. M. Rabbah. Lime: a java-compatible and synthesizable language for heterogeneous architectures. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, *OOPSLA*, pages 89–108. ACM, 2010.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *JPDC*, pages 207–216, 1995.

[5] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[6] P. Brauner and W. M. Taha. Globally parallel, locally sequential. Or, preserving natural parallelism. Report on early results. October 9th, 2010. Available from [1].

[7] J. Bruneau, C. Consel, M. O'Malley, W. Taha, and W. M. Hannourah. Preliminary Results in Virtual Testing for Smart Buildings (Poster). In *MOBIQUITOUS 2010, 7th International ICST Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, 2010.

[8] P. Charles, C. Grothoff, V. A. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In R. E. Johnson and R. P. Gabriel, editors, *OOPSLA*, pages 519–538. ACM, 2005.

[9] Y. Zhu, E. Westbrook, J. Inoue, A. Chapoutot, C. Salama, M. Peralta, T. Martin, W. Taha, M. O'Malley, R. Cartwright, A. Ames, and R. Bhattacharya. Mathematical equations as executable models of mechanical systems. In *First International Conference on Cyber-Physical Systems (ICCPS '10)*, 2010.