Numerical Algorithms with Tunable Parallelism

Aparna Chandramowlishwaran*, Abhinav Kahru†, Ketan Umare†, Richard Vuduc*
Georgia Institute of Technology
Computational Science and Engineering Division, College of Computing
266 Ferst Drive, Atlanta, Georgia 30332-0765, USA
*{aparna,richie}@cc.gatech.edu, †{akarhu3,kumare3}@gatech.edu

ABSTRACT

In this "idea" paper, we advocate the study of recently developed numerical algorithms that have *tunable parallelism*, which may provide a mechanism for a class of scientific applications to cope with or exploit heterogeneous multi- and many-core architectures. In particular, suppose we are given a physical system described by a partial differential equation (PDE) that we wish to integrate (*i.e.*, solve) numerically; rather than using traditional algorithms that are designed primarily with data-parallelism in mind, our goal is to produce numerical algorithms that have a mix of highly asynchronous task-level parallelism and synchronous data-parallelism, with an algorithmic "tuning knob" that at runtime can control the degree and type of parallelism. Such an algorithm and its implementation can in principle adapt to any hardware environment, whether it be highly multithreaded, support efficient data-parallel execution through large or small vector units, or have some heterogeneous mix of such components.

The mathematical framework we consider here is the *asynchronous variational integrator* (AVI) framework of Lew, Marsden, Ortiz, and West (2003), in which an interesting class of continuous physical systems can be modeled and solved using what are essentially *discrete-event simulation* techniques. Beyond having attractive numerical properties, the numerical algorithms derived in the AVI framework can have the sort of tunable-parallel property described above. We believe AVI-based algorithms constitute an interesting workload for evaluating emerging parallel hardware and software systems; by extension, AVI-like methods should be a fruitful area of collaborative algorithmic and systems research.

1. INTRODUCTION: WHY AVI?

The asynchronous variational integrator (AVI) framework of Lew, *et al.*, was originally developed for time-integration of partial differential equations (PDEs) arising in nonlinear elastodynamics, with the spatial domain discretized using finite elements [9]. The key numerical idea in the AVI framework is to allow each element of the spatial domain to have its own time step, as opposed to traditional time-integrators which use a uniform time step throughout the domain. In principle, an AVI costs many fewer flops than a traditional integrator, since the solution in each region of the domain "evolves" only as needed. (An AVI may execute two orders of magnitude fewer flops than a synchronous algorithm [7].) The geometric and numerical features of the solution at each element dictate the time step, and there are execution dependency constraints among elements to ensure causality. While there are other multiple time-step integrators in the literature, AVIs appear to have theoretically more clean and sound numerical properties [9]. For instance, AVIs are automatically symplectic, and time steps need not be integral multiples of each other.

AVIs are interesting from a systems and parallelism perspective because the asynchronous approach implies task-level parallelism, compared to a traditional integrator that is generally dominated by data-parallelism. Issues of dynamic scheduling, migration, and synchronization are of paramount importance for AVIs.

"Tunable parallelism" in AVIs arises as follows. The basic unit of work in an AVI is an element update, which involves a regular data-parallel computation. For each element, geometric properties dictate the maximum possible time step; we are free to aggregate elements to coarsen the unit of work, and then to choose the time-step to satisfy the upper-bounds for all aggregated elements, thereby obtaining an effectively larger unit of data-parallel work. In this way, we can effectively tune the degree and type of available parallelism.

We have thus far pursued three broad AVI implementation approaches for multicore systems, to better understand the inherent parallelism and asynchronous behavior of AVIs. The first is a traditional Pthreads implementation, with novel judicious scheduling of element updates. In the second, we recognize that AVIs are often implemented using shared data structures, the access to which may require heavy synchronization; we consider a software transactional memory-based implementation in this case. Thirdly, an AVI may be treated as a *discrete event simulation* (DES), and so we have implemented AVI within a parallel DES system, GTW [5]. We discuss this work below.

Figure 1 illustrates the asynchronous processing of mesh elements in a 4-thread run of one of our implementations. The domain is an elastic box that has been stretched to the right (in the positive x-direction) and has been discretized into approximately 200,000 triangular elements. The left subplot shows where in the domain the first 5,000 element updates occurred; each update is shaded by one of four colors, one color for each of the 4 threads. The right subplot shows the next 5,000 elements. We see clusters of updates occurring near elements with the smallest time-stamps, with regions updated irregularly, *e.g.*, compare the region inside the grey circle centered at approximately $(x, y) = (.45, .52)$.

2. PARALLELIZATION ISSUES

There have been at least two prior efforts to parallelize

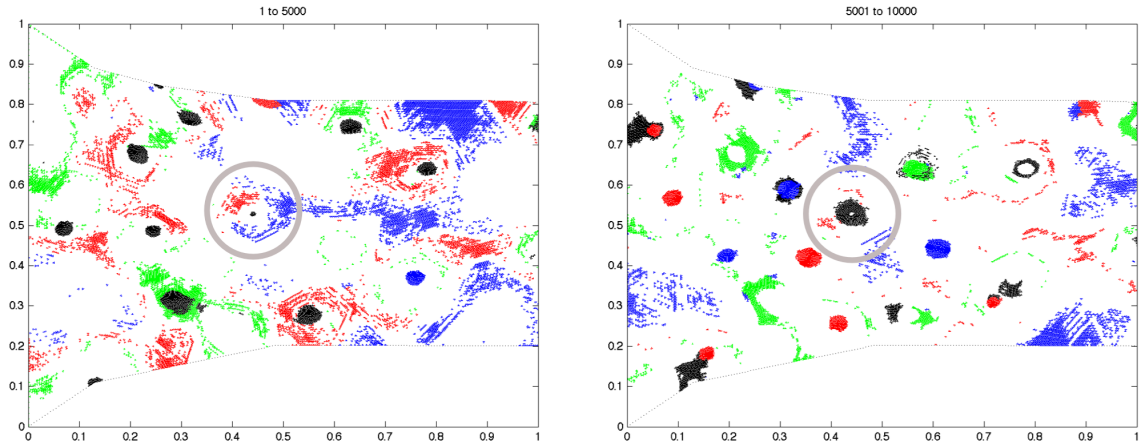

Figure 1: (Color) Two consecutive snapshots of active regions in a finite element mesh, for a 4-thread run. We color each element according to which of the 4 threads (red, green, blue, or black) is updating it.

AVIs, both of which partition the domain and assign subdomains to logical processes, for shared [6] and distributed [7] memory systems. Both rely on globally shared data structures, which can be synchronization bottlenecks.

Our work builds on the shared memory implementation of Huang, *et al.* [6]. Huang, *et al.*, first observe that the mesh elements, each with its own time-step (and, therefore, simulation time-stamp of its next update), implicitly define a dependency graph. Each mesh element is a node; there is an edge (u, v) if u and v are adjacent in the mesh and the current time-stamp of v is less than that of u (the causality constraint). They then show that, in the best case, there can be abundant parallelism: a mesh of n elements with an average of d neighbors each will have $O(\frac{n}{d})$ elements with only in-edges, *i.e.*, which are ready to execute because they are "local minima" in their time-stamp [6]. Their implementation consists of a global shared work queue of these "ready elements"; a free thread extracts a work unit from this queue and updates the corresponding element. After updating an element, the thread checks if completion of an update at the current element may enable a neigboring element to execute, and if so, moves that element into the ready queue. To increase the computation per work unit, we may group elements into larger "super-elements." The overall approach requires synchronization to globally shared data structures.

We have considered several alternative implementation choices and improvements, summarized as follows.

Smarter scheduling. A thread may choose any element in the work queue; some elements may actually enable more new ready elements if judiciously selected. We have considered a variety of approaches for identifying these elements based on inspection of neighboring time-stamps and dependency structures within the neighborhood of each element.

More efficient shared data structures. We have replaced the existing data structures with new, more efficient, and more modular data structures with finer-grained lock placement and better locality. In addition, we have considered traditional Pthreads-based implementations as well as implementations based on the Rochester Software Transactional Memory library [4].

Optimistic parallel discrete event simulation (DES) software frameworks. An AVI can be naturally cast as a DES: element (or super-element) updates become events with prescribed time-stamps, and the DES software framework takes care of all the scheduling and causality-preserving details. Indeed, implementing AVI in DES is relatively simple, requiring significantly fewer lines of code than building AVI from scratch. Moreover, we benefit from existing research and implementation in optimistic parallel DES, in which events execute optimistically and are rolled back automatically if causality constraints are violated, which is similar in spirit to transactional memory systems.

3. CURRENT STATUS

The combination of the techniques we have investigated has led to more scalable AVI implementations than in the prior work by Huang, *et al.* For instance, we have observed speedups near 30 using 64 threads on a Niagara 2 machine. Our current status is to test the tunable task- and data-parallel hypothesis by implementing aggregation for data-parallelism, where the data-parallel computations tuned with respect to the available SIMD or vector hardware, and for more irregular mesh discretizations than the one shown in Figure 1, which would better stress-test the methods.

Note that a parallel DES formulations of continuous systems simulations are not limited to AVI methods; we refer the interested reader to additional related work [8, 10].

AVIs are similar in spirit to recent work in which "classical" dense linear algebra kernels are reformulated to expose task-level parallelism [1, 2]. We may more broadly interpret the AVI implementation as generating a dynamically changing directed acyclic graph for which we seek efficient systems support for parallelization, reminiscent of work pursued in the by-gone era of dataflow architectures [3].

Acknowledgments

We wish to thank Richard Fujimoto and Jen-Chih Huang for pointing us in the direction of AVIs.

4. REFERENCES

[1] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. Technical Report LAWN 191,

UT-CS-07-600, University of Tennessee, September 2007.

[2] E. Chan, E. S. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn. SuperMatrix out-of-order scheduling of matrix operations for SMPand multi-core architectures. In *Proc. ACM SPAA*, pages 116–125, 2007.

[3] F. T. Chong, S. D. Sharma, E. A. Brewer, and J. Saltz. *Multiprocessor runtime support for fine-grained irregular DAGs*. Nova Science Publishers, Inc., 1995.

[4] L. Dalessandro, V. J. Marathe, M. F. Spear, and M. L. Scott. Capabilities and limitations of library-based software transactional memory in c++. In *Proc. TRANSACT*, Portland, OR, USA, August 2007.

[5] S. Das, R. M. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. GTW: A time warp system for shared memory processors. In *Proc. Winter Sim. Conf.*, pages 1332–1339, 1994.

[6] J.-C. Huang, X. Jiao, R. M. Fujimoto, and H. Zha. DAG-guided parallel asynchronous variational integrators with super-elements. In *Proc. Summer Comp. Sim. Conf.*, San Diego, CA, USA, July 2007.

[7] K. G. Kale and A. J. Lew. Parallel asynchronous variational integrators. *Int. J. Numer. Meth. Engng*, 70:291–321, 2007.

[8] H. Karimabadi, J. Driscoll, Y. A. Omelchenko, and N. Omidi. A new asychronous methodology for modeling of continuous systems: Breaking the curse of the Courant condition. *J. Comp. Phys.*, 205:755–775, 2005.

[9] A. Lew, J. Marsden, M. Ortiz, and M. West. Asynchronous variational integrators. *Arch. Rational Mech. Anal.*, 2003.

[10] J. J. Nutaro. *Parallel discrete event simulation with application to continuous systems*. PhD thesis, 2003.