PARALLEL ASYNCHRONOUS VARIATIONAL INTEGRATORS

Adrian Lew

Stanford University
Durand 207, Stanford, CA 94305-4040, USA
e-mail: lewa@stanford.edu

Keywords: multiple time steps, elastodynamics, variational integrators

Abstract. This paper presents a scalable parallel variational time integration algorithm for nonlinear elastodynamics with the distinguishing feature of allowing each element in the mesh to have a possibly different time step. Furthermore, the algorithm is obtained from a discrete variational principle, and hence it is termed Parallel Asynchronous Variational Integrator (PAVI). Based on a domain decomposition strategy, PAVI combines a careful scheduling of computations with fully asynchronous communications to provide a very efficient methodology for finite element models with even mild distributions of time step sizes. Numerical tests are shown to illustrate PAVI’s performance displaying excellent scalability properties. Additionally, a numerical example in which PAVI needs \( \approx 100 \) times less computing than an explicit synchronous algorithm is shown. This paper summarizes the ideas and results in [7].
1 INTRODUCTION

This paper presents a scalable parallel variational time–integration algorithm for non-linear elastodynamics. The algorithm builds upon the Asynchronous Variational Integrators (AVI) originally introduced in [12], and is therefore termed Parallel Asynchronous Variational Integrator (PAVI). This manuscript summarizes the ideas and results originally presented in [7], to which we refer the readers for further details.

Variational integrators constitute a novel paradigm for the construction of time–integration algorithms that, by design, guarantee a number of desirable and remarkable properties; see [1, 16, 17, 15, 8, 14, 11, 22, 3, 10, 13, 21, 12]. Chief among them are the existence of a discrete version of Noether’s theorem, the fact that every variational algorithm is automatically symplectic, and the consequent nearly perfect energy conservation properties for explicit algorithms at moderately large time steps. The distinguishing characteristic of variational integrators is that the equations that define an algorithm are the Euler–Lagrange equations of a discrete variational principle. In other words, the discrete motion is the stationary point of a functional over all discrete trajectories, termed the discrete action sum. The remarkable properties of the resulting algorithms can be directly traced back to their variational structure. These and other features of the theory of variational integrators have been thoroughly discussed in many of the previous references, in particular [11] or [15], and hence we shall skip further discussions herein.

Asynchronous Variational Integrators are a class of variational integrators that, in combination with finite element discretizations of an elastic medium, permit every element in the mesh to have a different time step. Furthermore, the time steps need not bear any integer multiple relation with each other, a distinguishing feature with respect to other multiple time step integrators, such as subcycling algorithms (e.g., [19]), element–by–element methods [6, 5], and the impulse method in molecular dynamics, also known as Verlet–I and r–Respa [4, 20].

The paper begins by briefly reviewing salient features of AVI in section 2, as needed in order to introduce PAVI in section 3. Section 4 demonstrates the scalability and advantage of the proposed algorithm by way of two numerical examples, one of which displays striking computational advantages of utilizing fully asynchronous schemes.

2 ASYNCHRONOUS VARIATIONAL INTEGRATORS

The formulation of AVIs has been comprehensively described in [12, 11]. We shall herein skip the details of the variational structure, and directly describe the resulting algorithm.

Consider a nonlinear elastic body discretized by a finite element mesh. For simplicity, we explain here the case in which all of the degrees of freedom are nodal displacements. Let \( x_a \) describe the spatial position of node \( a \) in the mesh, let \( m_a \) be its lumped mass, and let \( x_K \) denote the set of nodal positions whose associated basis function has support on element \( K \). The strain energy of the element \( V_K(x_K) \) is obtained as

\[
V_K(x_K) = \int_K W \left( \sum_{a \in K} x_a \nabla M_a(X) \right) \, dV,
\]  

(1)

where \( M_a(X) \) is the shape function associated with node \( a \) and \( W: \mathbb{R}^{3 \times 3} \rightarrow \mathbb{R} \) is the nonlinear strain energy density of the material.

The time discretization follows by choosing a time-step \( \Delta t_K \) for each element \( K \). For simplicity, we consider here \( \Delta t_K \) to be constant. The case with variable time steps per element is discussed in [10], in which numerical examples of elemental time steps changing as the simulations progresses are also shown. Consequently, herein the constitutive relation of element \( K \)
is computed only at times $t^j_K = j\Delta t_K$. Any time this computation is performed, the degrees of freedom in $x_K$ are updated according to the following scheme

$$p^{i+1/2}_a - p^{i-1/2}_a = I^i_a,$$  

(2)

where

$$p^{i-1/2}_a \equiv m_a \frac{x^i_a - x^{i-1}_a}{t^i_a - t^{i-1}_a} \equiv m_a v^{i-1/2}_a,$$  

(3)

and

$$I^j_{1_K} \equiv -(t^j_K - t^{j-1}_K) \frac{\partial}{\partial x^j_{1_K}} V_K(x^j_{1_K}).$$  

(4)

Here $x^j_{1_K}$ denotes the values of the degrees of freedom $x_K$ at time $t^j_K$. Since the degree of freedom $x_a$ of any given node $a$ generally belong to more than one set of elemental degrees of freedom $x_K$, the value of $x_a$ is modified more often that $x_K$. Let $x^j_a$ denote its value at time $t^j_K$, and we set $t^j_a = t^j_K$ as the time of the $i$-th time-step for node $a$. A similar convention is adopted for the values of the elemental and nodal impulses $I^j_{1_K}$ and $I^i_a$, respectively. Equations (2), (3) and (4) possess the distinguishing feature of being the (discrete) Euler–Lagrange equations of a discrete variational principle, see [11] for details.

The following simple one-dimensional example illustrates how serial AVIs are implemented in practice. Figure [1] shows a space–time diagram of the evolution of a one–dimensional elastic medium in time, computed with a 3–element mesh. At $t=0$ the initial positions $x^0_a$ and momenta $p^{1/2}_a$ for each node are assumed to be known, determined from the initial conditions of the problem. In this example element #3 has the smallest time step, followed by element #1 and then element #2. The computation begins with an elemental update for element #1. Equation (3) is used to obtain the value of $x^j_K$, $K = 3$, and the value of $I^j_{1_K}$ then follows from equation (4). An updated value for the momentum of each node in element #3 is obtained then from equation (2). At this point a new set of positions and momenta for each node in the mesh is available, and the update procedure can be repeated on any other element in the mesh. The computation continues...
by always updating the element with the next earliest update time. Hence, the sequence of
elemental updates for this example is #3, #1, #3, #2, #1, #3, #1, #3, #2. It is simple to check
that in this way all information needed to update each element is always readily available.
The scheduling of the elements is efficiently performed through a priority queue (PQ) [9].

The choice of the time step size for each element is determined by the application. Very
often, however, it is sufficient to choose it according to the local Courant condition, i.e., such
\( \Delta t = f \frac{h}{c} \), where \( h \) is the element size, \( c \) is a nominal dilatational wave speed in the material
and \( f \in (0, 1) \). Elemental time steps can of course be changed as the computation evolves.
However, multi–time step methods are prone to develop the so–called resonance instabilities
(e.g., [2, 18]), which may limit the range of available choices in practice.

3 PARALLEL ASYNCHRONOUS VARIATIONAL INTEGRATORS

3.1 When can elements be updated?

The formulation of the algorithm for AVIs described in section 2 is not ready to perform
efficiently in parallel computer environments. The essential reason is that using a unique pri-
ority queue to schedule elemental updates is an inherently sequential solution. The use of a
single priority queue is however not necessary; the asynchronous nature of AVI can be further
exploited to construct an efficient and scalable parallel implementation.

Consider initially the condition under which any elemental update can be performed. To this
end, assume that all elemental updates for any particular element \( K \) are performed in chrono-
logical order: if \( t_{iK}^j < t_{iK}^{j'} \), and hence \( j' > j \), the elemental update at time \( t_{iK}^{j'} \) is always performed
before the one at time \( t_{iK}^j \). The necessary and sufficient condition to conduct the update of ele-
mement \( K \) at time \( t_{iK}^j \) is that the information needed to solve equations (2)–(4) for \( x_{a}^{i} \) and \( p_{a}^{i+1/2} \)
be available, i.e., the values of \( x_{a}^{i-1} \), \( t_{a}^{i-1} \), \( p_{a}^{i-1/2} \) for all nodes \( a \in K \), and the value of \( t_{iK}^{j-1} \) are
required. By simple inspection it is evident that this condition is fulfilled whenever the next
elemental update time for any of the elements in \( N_K \) is larger or equal than \( t_{iK}^j \). Here \( N_K \) de-
notes the set of elements that share one or more degrees of freedom with \( K \), not including \( K \);
for finite element meshes these are the topological neighbors of element \( K \). Figure 2 illustrates
this condition with an example. We shall label this condition as the “coloring condition”, based
on an analogy of PAVI with a graph coloring problem detailed in [7].

The central idea behind the formulation of a parallel AVI algorithm is that at any time in the
computation there will often be many elements ready to be independently updated, not only the one with the highest priority next elemental update.

### 3.2 Formulation of PA VI

We illustrate next the formulation of PA VI with an example; we refer the reader to [7] for the specific details.

Consider a two–processor, four–element example, see figure 3. For PA VI we adopt a domain decomposition strategy, and distribute all elements in the mesh among processors. In our example elements #1 and #2 are assigned to processor $p_0$, and elements #3 and #4 to processor $p_1$. Two key new concepts introduced in PA VI are: a) one priority queue per processor, labeled $PQ_0$ and $PQ_1$ in the example and, b) a waiting list per processor, labeled $WL_0$ and $WL_1$. The interaction between the two can be roughly summarized as follows. The priority queue in each processor schedules the order of the elemental updates. However, an elemental update may be popped from the PQ that does not satisfy the coloring condition. This will happen any time some needed information for the elemental update is not available, either because the last elemental update for one of the surrounding elements could not be completed, or because the information has simply not yet arrived from another processor. In this case the aforementioned update is sent to the WL, and linked to the element whose information is missing. Once the missing information becomes available, the elemental updated is sent back to the PQ. A more detailed explanation is provided in [7].

The interaction between the PQ and WL is illustrated in figure 3, which shows a sequence of six consecutive steps. The evolutions of the contents of the PQ and WL for each of the two processors are shown besides and at the bottom of each processors elements, respectively. Displayed are only the element numbers associated with the elemental updates in them. Note that, as aforementioned, each WL has a list associated with elements in other processors, in this case $WL_0(3)$ and $WL_1(2)$. At each step the edges of the graph emerging from already updated vertices are drawn with thin lines, while the others are represented by way of dashed lines. For this example we assume that once information is computed it can be instantaneously communicated to the other processor, and that both processors take a constant time to perform whatever tasks are needed between two consecutive extractions of element updates from the priority queue, regardless of whether the element is updated or not.

Step 1 in figure 3(a) shows the status of both PQs after initialization before any element has been popped. At this stage both waiting lists are empty. As shown in step 2, figure 3(b), processor $p_0$ then pops the topmost element update from $PQ_0$, (1, 1), checks it satisfies the coloring condition and performs the update, as indicated with a circular ring. Processor 1 proceeds to pop the elemental update (3, 1) from $PQ_1$, but finds that it does not satisfy the coloring condition because elemental update (2, 1) is not yet available. Elemental update (3, 1) is consequently sent to $WL_0(2)$, as shown. No information is exchanged between processors at this point. During step 3, figure 3(c) processor 0 pops elemental update (2, 1), at the top of the priority queue, and since it satisfies the coloring condition it is performed. This information is then sent to processor 1, which has in the meantime unsuccessfully attempted to perform elemental update (4, 1), and therefore redirected it to $WL_1(3)$. At this point $PQ_0$ has elemental update (1, 2) at its top, while $PQ_1$ is empty. So far little progress has occurred in processor $p_1$. At the beginning of step 4, figure 3(d) processor 1 receives the information about elemental update (2, 1) and empties $WL_1(2)$. It then sends elemental update (3, 1) to the top of $PQ_1$, from where it is popped and performed. The last novel feature is found during step 5, figure 3(e) since processor 0 first receives the information about the elemental update (3, 1), and when it pops elemental update...
Figure 3: Step-by-step example of PAVI with two processors and four elements. Solid lines in the graph indicate dependencies for which information is already available. Information across processors is assumed to travel instantaneously. Each snapshot shows the state of the algorithm after an element has been popped from the PQ and a decision has been made about updating it (circled) or not (sent to WL).
(a) Synchronous algorithm

(b) Asynchronous algorithm

Figure 4: One-dimensional, six-element example of the partition of a mesh in the synchronous and asynchronous cases. Naturally, the synchronous case assigns the same number of elements to each processor. In contrast, the asynchronous one also assigns roughly the same number of elemental updates to each processor, achieved only by way of a different number of elements in each subdomain.

(2, 2) from PQ0 it is able to carry it out.

### 3.3 Mail system and load balancing

When synchronous explicit algorithms for elastodynamics are adopted the load balancing is traditionally achieved by virtue of partitioning the mesh into groups with roughly the same number of elements, and such that the number of shared degrees of freedom between processors is minimized. The analysis is different with asynchronous algorithms, since every element is entrusted with a different computational load. Elements having smaller time steps need to be updated more often than elements with larger time steps, and hence require a larger computational load. The domain decomposition strategy should be sensitive to this fact; the computational load of each element is inversely proportional to its time-step, and the elements should be distributed such that the sum of the loads of all elements in each processor is roughly the same, see figure 4.

One of the advantages of the fully asynchronous approach is that there is rarely a need for a processor to wait idle for information to arrive, since most often there are elemental updates that already satisfy the coloring condition. The communication strategy needs to be designed to take advantage of this feature.

When a message passing interface paradigm for communications is adopted, an appealing communication strategy can be constructed by mimicking ideas from the day-to-day postal system; we call it the *mail system*. In this context, processors receive and send information as mail packets. Any time a processor has new information relevant to elemental updates in other processors, it sends it in a message to each processor that needs it, the *mail*. On the other end, each processor knows that it has to periodically check its mailbox for new incoming messages. The distinguishing feature of this approach is that neither the sending processor has to wait for the message to be delivered before moving on to other tasks, nor the receiving processors has to remain idle waiting for a given message to arrive. In fact, the receiving processor does not know what message it is expecting, it only knows about it after opening the mail. This is in contrast with most synchronous strategies in which the receiving processor knows what information the
(a) Geometry of the test case, which represents a crude model of a turbine rotor with 32 blades.

(b) Partial view of the mesh. It contains 18724 ten–nodded tetrahedral elements and 47517 nodes.

(c) Partitions of the mesh in the case of 4 processors. Notice the load balance and minimized edge-cut.

(d) Speed-up on a fast myrinet network

Figure 5: Geometry, mesh and scalability.

incoming message should contain.

By utilizing the mail system, the communication frequency generated by two neighboring elements in different processors is tightly related to their time steps. Elements with smaller time steps will get updated more often and hence would have to send information more frequently. This needs to be accounted for when partitioning the problem, to minimize the amount of interprocessor messages needed.

The decomposition of the domain is the accomplished by solving a graph partitioning problem. Each element is represented by a node in the graph, and two nodes are joined by an edge whenever the two elements are neighbors. The weight of each node is inversely proportional to the time step of the element, and the weight of an edge is equal to the maximum weight among the two nodes it joins. The goal is then to partition the nodes into groups of equal weights by minimizing the sum of the weights of edges stranding nodes in different partitions.
Figure 6: Mesh and partitions used in the AFM cantilever simulation. The geometry represents a typical cantilever used in tapping mode. The mesh contains 14616 elements and 24908 nodes. The smallest elements are of the order of $10^{-8} \text{m}$ in size, while the length of the cantilever is 120 $\mu\text{m}$. Remarkably, the small elements near the tip shift all partition boundaries towards it.

4 PERFORMANCE EVALUATION AND NUMERICAL EXAMPLES

We show next two numerical examples of PAVI. In the first example we demonstrate the scalability of the proposed algorithm, while the second shows a striking performance improvement of PAVI with respect to a synchronous counterpart. More examples with a more extensive discussion is provided in [7].

4.1 Scalability study

The geometry of the problem under consideration is shown in figure 5(a). It represents a crude model of a turbine rotor with 32 blades. A section of the associated ten–nodded tetrahedral mesh is shown in 5(b). The mesh is essentially invariant when rotated around the turbine axis by $2\pi/32$. It contains 18724 elements and 47517 nodes. The material of the mesh is a nonlinear elastic neo-hookean model extended to the compressible range (see, e.g., [11]).

The problem solved consists on imposing an initial angular velocity around the turbine axis on the initially stress–free and static structure. This problem was repeatedly solved with exactly the same mesh in different number of processors. As an illustration of the load–balancing strategy, the partitions in case of 4 processors are shown in figure 5(c). As expected, the do-
mains are well balanced in terms of computational load because of the symmetry of the mesh. Also, notice that the domain boundaries are created between the blades, thus minimizing the communication costs.

Scalability results in the form of a speed-up curve are presented in figure 3.3 with solutions on $P = 2^i$ processors, $i = 0, \ldots, 7$ processors. These results were obtained on a fast myrinet network which has a very low latency. The algorithm shows the superlinear speed–up up to 32 processors, because of the decreasing cost of the PQ. Beyond 32 processors, the scalability starts to decline slightly, reflecting both the effects of the change in domain partition patterns (the turbine has only 32 blades), as well as the relative increase in amount of communications and decrease in amount of computations in each processor. This onset of performance decline is always expected for this type of scalability studies. For a problem with a fixed size there is generally an optimal number of processors, since any advantage gained from the parallel processing is effaced by the increasing relative cost of communication versus computations. It is natural, however, to wonder whether competing synchronous algorithm also see a performance degradation beyond 32 processors; this is in fact the case, and it is shown in [7].

4.2 Numerical Example

We present next a practical example for which PAVI is an ideal choice. This is the case for the simulation of the dynamics of an AFM cantilever that is excited at a frequency close to its natural one. The mesh for an idealized AFM silicon cantilever is shown in figure 6(a). It consists of 14616 ten–nodded tetrahedral elements and 24908 nodes. The length of the cantilever is 120 $\mu$m, and its conical tip is capped by a spherical section of radius 4 $\mu$m, a very blunt tip.

The simulation starts from a stress–free configuration initially at rest. The simulation was performed on 4 processors on a gigabit ethernet network. The domain was partitioned into 4 subdomains, one for each processor, which are shown in figure 6(b). As expected, the computa-
tional cost of the elements near the tip is very high due to their small time steps. Consequently, the partition boundaries are concentrated in the tip region.

The full exploitation of the multiple time scales in PAVI is on display in figure 7(a) which shows a contour plot of the total number of elemental updates for each element in a region of the mesh near the tip. The maximum and minimum time steps in the mesh are $662 \times 10^{-7} \mu s$ and $1.47 \times 10^{-7} \mu s$, respectively. Consistent with these values, the number of elemental updates in the contour plot also span two orders of magnitude. It is enlightening to see how many elements with very small time steps there are in the mesh. This is shown in figure 7(b) which displays only those elements that have had more than $10^{5.8}$ elemental updates throughout the simulation. Since these are relatively few, PAVI is positioned to deliver an unmatched performance by any other synchronous methods. In fact, had this simulation been computed with a parallel synchronous algorithm, a total of $260 \times 10^9$ elemental updates would have been needed. Instead, with asynchronous updates only $3.3 \times 10^9$ elemental updates are necessary; this is two orders of magnitude less computing!

REFERENCES


