14th World Congress on Computational Mechanics (WCCM)
ECCOMAS Congress 2020
Virtual Congress: 11–15 January 2021
F. Chinesta, R. Abgrall, O. Allix and M. Kaliske (Eds)

HIERARCHICAL PARALLELISM FOR TRANSIENT SOLID MECHANICS SIMULATIONS

DAVID J. LITTLEWOOD[1], REESE E. JONES[2], NICOLAS M. MORALES[3], JULIA A. PLEWS[4], ULRICH HETMANIUK[5], AND JONATHAN J. LIFFLANDER[6]

[1] Sandia National Laboratories
PO Box 5800, MS 1322, Albuquerque, NM 87185
djlittl@sandia.gov

[2] Sandia National Laboratories
PO Box 969, MS 9161, Livermore, CA 94551
rjones@sandia.gov

[3] Sandia National Laboratories
PO Box 969, MS 9158, Livermore, CA 94551
nmmoral@sandia.gov

[4] Sandia National Laboratories
PO Box 5800, MS 0845, Albuquerque, NM 87185
japlews@sandia.gov

[5] NexGen Analytics
412 N Main Street, Suite 100, Buffalo, WY 82834-1761
ulrich.hetmaniuk@ng-analytics.com

[6] Sandia National Laboratories
PO Box 969, MS 9158, Livermore, CA 94551
jliffla@sandia.gov

Key words: Solid Mechanics, Contact, Finite Elements, Asynchronous Task Scheduling, GPUs

Abstract. Software development for high-performance scientific computing continues to evolve in response to increased parallelism and the advent of on-node accelerators, in particular GPUs. While these hardware advancements have the potential to significantly reduce turnaround times, they also present implementation and design challenges for engineering codes. We investigate the use of two strategies to mitigate these challenges: the *Kokkos* library for performance portability across disparate architectures, and the DARMA/***vt*** library for asynchronous many-task scheduling. We investigate the application of *Kokkos* within the *NimbleSM* finite element code and the LAMÉ constitutive model library. We explore the performance of DARMA/***vt*** applied to *NimbleSM* contact mechanics algorithms. Software engineering strategies are discussed, followed by performance analyses of relevant solid mechanics simulations which demonstrate the promise of *Kokkos* and DARMA/***vt*** for accelerated engineering simulators.

1 INTRODUCTION

Recent advances in hardware for massively parallel scientific computing have yielded performance improvements through increased parallelism thanks to on-node accelerators, especially graphics processing units (GPUs). Hierarchical parallelism, with one level of parallelism occurring between compute nodes and the second level on-node via the GPU, is commonplace. These novel technologies enable the simulation of large computational models and improve the outlook for once intractably expensive high-fidelity approaches. Increasingly complex architectures, however, necessitate careful software engineering. Principal concerns include full utilization of GPU computational power, memory management, task scheduling, load balancing, developer productivity, and long-term maintainability of source code.

In this work, we present two ongoing efforts to improve the design and implementation of finite element analysis codes suitable for next-generation platforms. The first is the *Kokkos* software library for performance portability [2]. *Kokkos* strives to provide an abstract API for critical programming elements such as multi-dimensional arrays and parallel looping constructs. Code written to the *Kokkos* API is translated at compile time to a hardware-specific backend, enabling deployment of the same software across a variety of architectures. The second strategy is asynchronous many-task (AMT) scheduling using the DARMA/***vt*** library. AMT is an alternative to procedural, MPI-based approaches in which programs are defined in terms of tasks and data. Developers create tasks with well-defined interdependencies and data structures utilized by these tasks. At run time, the AMT scheduler manages the execution of work for available resources. This may include executing tasks asynchronously, or moving data among hardware resources to optimize task execution.

We apply *Kokkos* and DARMA/***vt*** to the *NimbleSM* C++ Lagrangian finite element code for explicit transient dynamics to optimize performance and improve software developer productivity. We first apply *Kokkos* to create a framework within *NimbleSM* in which the majority of data is managed with `Kokkos::View` containers, and computationally expensive kernels are executed using the `Kokkos::parallel_for` construct. We then focus on the use of *Kokkos* for constitutive model development, since constitutive models are among the most complex, computationally expensive, and difficult-to-maintain routines in a FEM code. We define a constitutive model interface that utilizes *Kokkos* for improved performance while maintaining a design that is familiar to the applied mechanics community. Principal considerations include memory allocation, data transfer, and support for virtual object hierarchies. Lastly, we apply the DARMA/***vt*** library to improve the parallel scaling of contact mechanics algorithms. We focus on contact because its computational expense evolves dynamically over the course of a simulation, making it highly susceptible to load imbalance. Furthermore, kernels for contact mechanics are typically modular relative to the overall FEM code, making them amenable to encapsulation and AMT management. The DARMA/***vt*** library enables asynchronous one-sided communication, virtualization, and dynamic load balancing for collision detection and contact enforcement, but comes at the cost of increased overhead due to task management. The overall software design for *NimbleSM* is illustrated in Algorithm 1, in which the most computationally expensive routines for calculation of the internal forces are implemented with *Kokkos*, and routines specific to contact are implemented with DARMA/***vt***.

Algorithm 1 Pseudocode for *NimbleSM* explicit transient dynamics.

```
for each time step $n$ do
    Increment time $t^{n+\frac{1}{2}} \leftarrow \frac{1}{2}(t^n + t^{n+1})$ and $t^{n+1} \leftarrow t^n + \Delta t$
    Update velocity $\mathbf{v}^{n+\frac{1}{2}} \leftarrow \mathbf{v}^n + (t^{n+\frac{1}{2}} - t^n)\mathbf{a}^n$
    for each d.o.f. $i$ with a kinematic boundary condition do $v_i^{n+\frac{1}{2}} \leftarrow$ prescribed value
    Update displacement $\mathbf{u}^{n+1} \leftarrow \mathbf{u}^n + \mathbf{v}^{n+\frac{1}{2}}\Delta t$
    ▷ Compute internal forces
        Kokkos element.ComputeDeformationGradients()
        Kokkos material_model.ComputeStress()
        Kokkos element.ComputeNodalForces()
    ▷ Sum internal forces at MPI partition boundaries
        parallel_communicator.VectorReduction(internal_force)
    ▷ Compute contact forces
        DARMA/vt contact.ProximitySearch()
        DARMA/vt contact.Enforcement()
    ▷ Communicate contact forces and sum with internal forces
        parallel_communicator.CommunicateContactForces()
    Compute acceleration $\mathbf{a}^{n+1} \leftarrow \mathbf{M}^{-1}\boldsymbol{f}^{n+1}$
    Update velocity $\mathbf{v}^{n+1} \leftarrow \mathbf{v}^{n+\frac{1}{2}} + (t^{n+1} - t^{n+\frac{1}{2}})\mathbf{a}^{n+1}$
    if designated output step : io_system.WriteToFile()
end for
```

2 PERFORMANCE PORTABILITY WITH KOKKOS

Kokkos is a C++ library for performance portability across disparate hardware architectures [2]. It provides application developers with a well-defined, stable API for a number of programming elements, most importantly the `Kokkos::View` for data management, and programming constructs for parallelization such as `Kokkos::parallel_for`. The power of *Kokkos* lies in its ability to map code from the abstract API to an optimized, hardware-specific backend at compile time. In the case of data structures, *Kokkos* maps multi-dimensional arrays defined by the application developer to specific memory spaces and layouts for a given platform. When required, *Kokkos* provides the necessary utility functions to move data between devices (*e.g.*, CPU and GPU memory spaces). In the case of a `Kokkos::parallel_for`, *Kokkos* maps application code to an execution model specific to the hardware: C-style `for` loops on serial CPUs, OpenMP for hardware with multiple threads, or CUDA code in the case of GPUs, among others.

Our initial exploration of *Kokkos* for FEM solid mechanics codes focused on efficient utilization of GPUs for explicit dynamics. As illustrated in Algorithm 1, code execution for explicit dynamics is dictated by the velocity Verlet time integration scheme. Explicit dynamics simulations are comprised of a large number of small time steps that do not require the solution of a global system of equations. The computational expense of explicit dynamics is dominated by a small number of element and material-point routines, for example the calculation of the deformation gradient and evaluation of the constitutive model (refer to *Kokkos* annotations in Algorithm 1). To maximize performance, these routines were written to the *Kokkos* API such that they can be translated to CUDA at compile time and executed on GPUs at run time. An additional critical factor is data management. Our strategy was to utilize *Kokkos* data structures such that all node, element, and material-point data are stored in *device* (GPU) memory space and copied to *host* (CPU) memory space only when absolutely necessary, *e.g.*, for communication between MPI partitions or when writing output to disk.

Performance improvements resulting from the use of GPUs were investigated with the simulation of wave propagation in a notched plate shown in Figure 1. The simulation utilized roughly five million

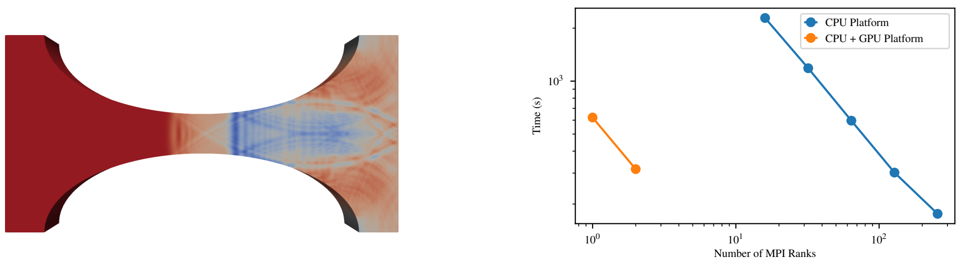

Figure 1: Wave propagation simulation utilizing *Kokkos* in the *NimbleSM* code. (left) Illustration of the velocity field, (right) scaling results with and without GPU accelerators.

fully-integrated hexahedral elements and a neo-Hookean material model. A uniform initial velocity was assigned to the entire domain, and a fixed displacement boundary condition was applied to the leading end of the plate, resulting in a compression wave. The simulation was run using a range of MPI ranks on two computing platforms. The first platform is a standard x86 compute cluster, and the second is a testbed equipped with two Intel Tesla P100 GPUs. Programming to the *Kokkos* API enabled code portability; the same *NimbleSM* code was used in both cases, but in the case of the GPU-equipped machine the relevant program elements were compiled with a CUDA backend. Figure 1 illustrates the corresponding performance improvements. Run times on the CPU-only platform are shown for 16, 32, 54, 128, and 256 ranks. In the case of the GPU-equipped testbed, run times are shown for 1 and 2 ranks, with each rank having sole access to a GPU. While *NimbleSM* showed excellent scaling in both cases, the overall run times differed by roughly 50$\times$, *i.e.*, matching the run time of a single rank with a single GPU on the testbed required roughly 50 ranks on the CPU-only cluster.

2.1 Application of Kokkos to Constitutive Models

The promising results obtained using *Kokkos* in *NimbleSM* for the wave propagation simulation motivated an in-depth study of the application of *Kokkos* to constitutive models. As discussed in Section 1, constitutive models represent a large fraction of the overall computational cost in solid mechanics codes.

There are several important characteristics of constitutive models that must be taken into account in the application interface design. The development of constitutive models requires particular subject matter expertise, leading to a natural division among code teams, so a well-defined, simple *Kokkos*-compatible software interface for the development of constitutive models is desirable. Yet the interface and associated data structures must be carefully designed to preserve performance. Additionally, in a given simulation a variety of constitutive models may be applied to specific subregions of the domain, motivating the use of dynamic polymorphism which is a practical software development strategy to maximize code reuse that has proven historically challenging on architectures such as GPUs [8].

The Library of Advanced Materials for Engineering (LAMÉ), a constitutive model library developed under the umbrella of the *Sierra/SM* nonlinear finite element analysis package [10, 11], attempts to address these considerations. LAMÉ is currently undergoing active development and refactoring to enable execution in a platform-portable way, yet with minimal disruption to the hundreds of legacy models.

2.1.1 Interface design for portability and productivity

Writing portable algorithms with the *Kokkos* package is straightforward. Eking out GPU performance, however, may require deep computer science understanding of data structure layouts, parallel looping constructs, and discrete memory spaces, implemented via templates or traits specified at compile time. Interfaces to material models are typically well defined, and the models themselves are developed by engineers less experienced in computer science and more experienced in mechanics. For this reason, we seek a material model interface that is free of *Kokkos*-specific constructs to support a simple process for porting a constitutive model to a GPU platform and allow developers to focus on the physics and core calculations of the constitutive model.

Although the desired programming interface is simple from the perspective of the constitutive model developer, *NimbleSM* and other FE simulators that execute these models organize their data in parallel, *Kokkos*-enabled data structures. Through *Kokkos*, *NimbleSM* and *Sierra/SM* prescribe `Kokkos::View` data array layouts in memory. Management of such data structures between application and *Kokkos*-indifferent library adds design complexity. In particular, a lack of attention to memory access patterns and data duplication may cause severe disparities in performance across heterogeneous platforms, especially GPUs.

The LAMÉ library adopts a flexible interface with no explicit notion of data layout in memory, nor explicit invocation of *Kokkos* constructs, instead relying on specially designed C++ types, such as scalars, tensors, and vectors, to *access* quantities for material-level computation, rather than *copy* into legacy pointer-based, layout-specific data structures. The material interface data access pattern is designed to encapsulate data layout dictated by the mechanics code (in this case *NimbleSM*) and seamlessly translate it to tensor $T(i,j)$, vector $v(i)$, or scalar s quantity comprehension.

To collect performance data on a hybrid CPU–GPU architecture, we employed a large finite element simulation with a neo-Hookean elastic model with 8-node, fully-integrated hexahedral elements. The problem had 4.32×10^6 elements and 5.29×10^6 nodes, of which 7.83×10^4 elements and 15.65×10^4 nodes were on the GPU-accelerated portion of the machine. Figure 2a demonstrates the difference in GPU performance between an implementation which creates temporary data copies for coalesced, pointer-style access for material-level calculations versus one in which field data access interface objects (tensor, vector, scalar comprehension) are used. Internal force computation includes material stress calculation, which is approximately equal in either case, while other costs, including data duplication, are considerably higher in the copy case. Eliminating data duplication and working on *in situ* field data leads to a 23% reduction in simulation runtime.

2.1.2 Polymorphism and data synchronization with *Kokkos*

Constitutive models in solid mechanics problems present an opportunity to exploit dynamic or runtime polymorphism. However, heterogeneous architectures such as GPUs complicate the execution of virtual functions, while standard data structures for managing polymorphic types, such as smart pointers, similarly are not conducive to use on GPUs. Thus, we employ a simple *Kokkos*-compatible pointer to manage allocation of objects in heterogeneous virtual memory. This entails both (a) allocation of space on the GPU for the object, and (b) destruction and freeing of allocated memory at the appropriate time.

In general material models may utilize *multiple levels* of virtual object hierarchies, for instance, incorpo-

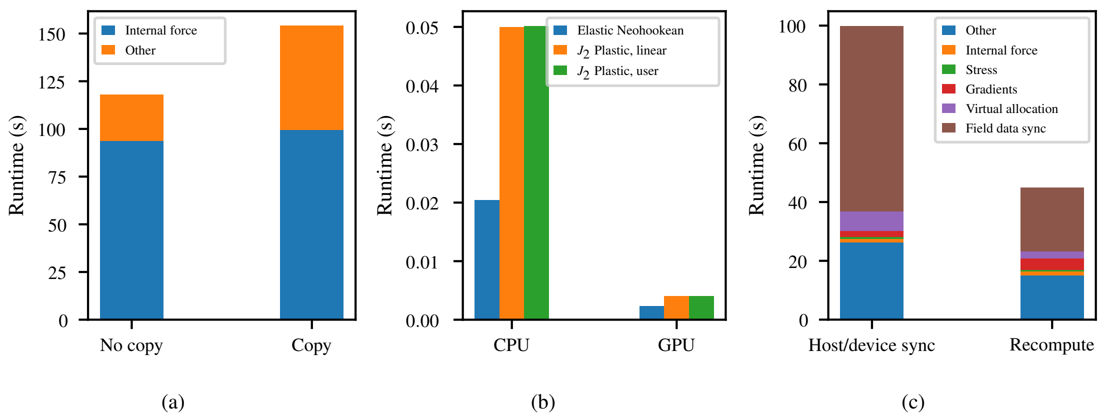

Figure 2: LAMÉ performance results. (a) Overall simulation timings: copying data into the material interface vs. accessing data without temporary copy. (b) Material evaluation cost through a virtual interface. 1,024,000 total material points; CPU data collected on 4 MPI ranks $\times$ 4 OpenMP threads, Intel Xeon Gold 6130; GPU data collected on one Nvidia Volta 100. (c) Overall simulation timings: field synchronization and memory reallocation for virtual objects vs. recomputation to alleviate synchronization cost and consolidated allocation of virtual objects.

rating user-defined routines to advance state data and prescribe behavior under given loading conditions. Figure 2b demonstrates standalone *Kokkos* LAMÉ model execution on both CPU MPI+OpenMP and GPU execution environments comparing three different models: an elastic model (incorporating only one virtual function call and no material state information), a J_2 plasticity model with linear hardening behavior (one virtual call, with material state information), and a J_2 plasticity model with a user-defined hardening model (additional virtual function calls for user-defined hardening, with material state information). The GPU execution represents a speedup of about 10$\times$ relative to CPU execution on the compute node, while the relative cost of materials with or without state data remains fixed between CPU and GPU execution. On the other hand, the additional virtual function call required in the user-defined J_2 plastic model incurs almost no computational overhead relative to the simple linear J_2 plastic model.

Because GPU memory allocation is quite slow relative to computation, memory allocated on the GPU can be reutilized when polymorphic objects need to be synchronized or updated during the course of a simulation. Using preallocated memory pools in GPU accelerator memory via the `Kokkos::kokkos_malloc<>()` routine, virtual objects may be reinstantiated as needed, finessing the cost of destroying then reallocating memory. Another complexity of working with GPUs is the explicit synchronization of *host* (CPU) and *device* (GPU or other accelerators) memory. This cost is exacerbated in constitutive model computations, which in general require large amounts of data at each integration point in a finite element mesh, such as state data and gradients of nodal fields. However, significant cost savings can be achieved by judiciously recomputing fields.

Performance improvements from leveraging recalculation of material point quantities to avoid synchronization and reinstantiation of objects in preallocated memory are demonstrated in Figure 2c. Synchronizing the minimal subset of data between CPU and GPU in this example problem results in a 65% reduction in field synchronization costs, while reuse of preallocated memory for virtual objects on the GPU similarly decreases the cost of overall object allocation by 64%.

3 ASYNCHRONOUS MANY-TASK SCHEDULING WITH DARMA/*vt*

While *Kokkos* enables the utilization of on-node accelerators for evaluation of computationally expensive kernels, it does not address the complementary issue of overall resource management on heterogeneous architectures. A prime example is load balancing. *Kokkos* allows units of work to be executed efficiently on a GPU, but does not ensure that network communication, CPU calculations, and GPU calculations are coordinated effectively. In a worst-case scenario, work may be assigned to a small set of compute resources, while other resources sit idle. Figure 3 shows an example of how load imbalance can have a negative impact on performance. In this figure, almost all ranks are idle, while one rank performs most of the work. Ideally, work should be distributed evenly among the ranks.

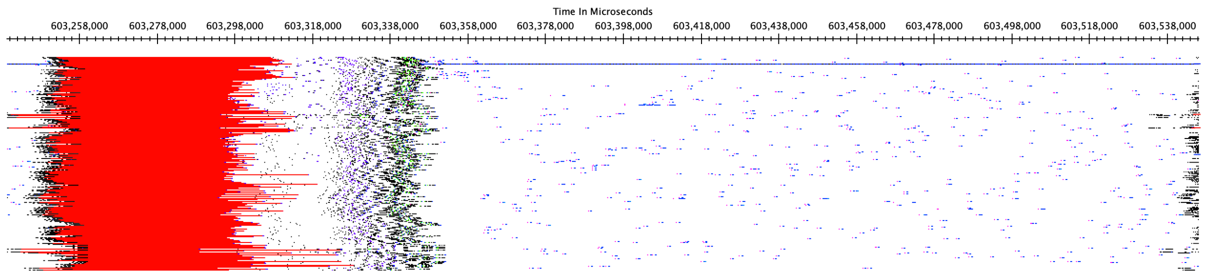

Figure 3: Resource utilization for a simulation exhibiting isolated contact. Blue lines indicate contact computations, while red represents non-contact portions of the simulation. Each rank is a separate entry on the y-axis.

AMT (asynchronous many-task) scheduling is a programming model and runtime framework that aims to improve resource utilization and developer productivity. This is achieved by defining programs, or elements of programs, purely in terms of tasks and data. In this way, application developers are freed to focus on the development of tasks, for example the software implementation of a physics model, while aspects of the code such as global data management and parallel communication are the responsibility of the AMT runtime.

We performed a preliminary investigation of the application of the DARMA/*vt* AMT library to contact mechanics in *NimbleSM*. Contact mechanics are a natural target for AMT scheduling because the expense of contact can vary wildly over the course of a simulation and is often isolated to small regions of the computational domain. Unlike the evaluation of material-point quantities such as stress, evaluation of contact forces do not load balance with static spatial or material decompositions. The primary drawback of bulk-synchronous models is their lack of dynamism and adaptability to conditions varying across time steps, difficulty in addressing different computational load configurations (such as the optimal computational load distribution among MPI ranks of contact and the remainder of the mechanics), and difficulty in implementation of one-sided communication and asynchronous tasks. These downsides can be remedied through the use of distributed asynchronous multi-tasking models. Distributed AMT models are available in programming models and libraries such as Charm++, Legion, and DARMA/*vt* [4, 9, 1, 7]. Examples of previous work for distributed AMT applied to contact include work by Ni, *et al.*, [9] with initial work using asynchronous data structures done by Harmon, *et al.* [3]. In the separate field of bulk-synchronous performance-portable contact, Lebrun-Grandié, *et al.*, [6] have developed the open-source *ArborX*. Their focus is on both efficient on-node and off-node parallelism with *Kokkos* and MPI.

3.1 DARMA/*vt* Background

DARMA/***vt*** is a framework for AMT that supports asynchronous one-sided communication through the use of messages and handlers. Messages can be addressed to an individual rank via the `sendMsg` command, causing the handler to be executed on the destination rank. Similarly, broadcast and reduce operations are also supported. Another abstraction that DARMA/***vt*** supports is is multi-dimensional, sparse or dense indexable collections that are overdecomposed over MPI ranks. With this abstraction, physics domains can be decomposed into multi-dimensional collections that naturally break down the simulated domain. The runtime system manages the mapping of collection elements to MPI ranks which it can migrate at runtime to improve the load distribution. With the collection API, users can call `sendMsg` to a particular collection element index causing a handler to be executed on the rank where the collection element currently resides. The system manages a distributed table and caching mechanisms to efficiently deliver messages to these overdecomposed entities.

This approach is advantageous for several reasons. First, the mapping (and remapping) of collection elements to ranks is handled by the runtime rather than the user, simplifying the development of scalable codes. This leads to advantages such as automatic instrumentation-based load balancing where the runtime can monitor tasks being executed by collection elements and use this database to apply a variety of highly scalable load-balancing techniques that migrate collection elements dynamically to improve execution time and reduce communication across ranks. Second, writing a problem in terms of overdecomposition provides a more natural mechanism to represent the problem domain, which provides performance portability; instead of being locked into a bulk synchronous, MPI rank-decomposed program representation, overdecomposition gives the runtime system the flexibility to rearrange execution depending on architectural characteristics or even runtime aberrations, such as a hard failure. Furthermore, representing the domain with finer-grained tasks enables the runtime scheduler to tune the ordering of task execution giving the runtime greater control over application performance. Finally, DARMA/***vt*** supports co-scheduling with MPI communication and the migration of per-rank data structures to overdecomposed collections. This allows one portion of a code to use traditional MPI methods for communication and computation, then switch to a distributed AMT approach using DARMA/***vt***.

3.2 Application of DARMA/*vt* to Contact Mechanics

A particularly challenging area for traditional bulk-synchronous programming models in solid mechanics simulations is collision detection (CD) and the element ghosting required for the modeling of contact. There are two main difficulties that traditional programming models encounter.

First, the collision detection problem is in itself dynamic. Distributed hierarchical CD approaches usually use preliminary stages of culling to determine which elements should be migrated/*ghosted* onto which MPI ranks. Later stages then perform a detailed search on-rank. However, this assignment of elements to ranks is not possible to determine in advance (although there are methods of near-time prediction using results from previous time steps). Bulk-synchronous approaches then require synchronization between ranks to arrange for a communication of elements. Asynchronous one-sided communication approaches, such as that found in DARMA/***vt***, on the other hand can send a message as soon as the early culling is complete. The asynchronous nature of this message allows the destination rank to perform work (such as handling other messages or performing early culling) without waiting for and synchronizing with the sending rank. This advantage becomes more noticeable with increased overdecomposition factors. If a collection of contact entities (called a *patch*) are subdivided into smaller patches, the actual work done per task becomes more fine-grained. This allows improved utilization of ranks processing search tasks, as the latency before a rank fires off its ghosting messages is reduced.

The second shortcoming of bulk-synchronous models is the difficulty of distributing load across ranks. A fundamental property of contact problems is that the required work is proportional to the number of contact interactions. However, the concentration of contact points can be unrelated to the decomposition of elements for the non-contact portion of the simulation. This can cause severe load imbalance where some ranks are stalled, waiting for busier ranks to complete their calculations. In some extreme instances, load imbalance can be so severe as to exceed the on-node memory of a rank, causing program termination. Therefore, it is important that compute models for contact be able to adjust computational load with ease. DARMA/***vt*** is particularly useful for this task, given the migratable nature of a collection of elements. Additionally, using asynchronous tasking, memory overloading problems can be avoided. If a patch is migrated to a rank, it can be discarded after the particular contact task it is associated with is completed. This technique works since the granularity of the problem is per-task. In short, AMT techniques such as DARMA/***vt*** can provide advantages when overcoming many of the computational challenges associated with contact. Additionally, the MPI interoperability of DARMA/***vt*** allows us to switch between MPI and tasking frameworks according to the work being performed.

The DARMA/***vt*** implementation for contact operates in four phases: tree-building/bounding box construction, distributed broadphase, midphase, and narrowphase closest-point projection with enforcement. Tree-building and bounding box construction calculate the overall bounding box of each patch. The bounding box itself is a configurable k-axis discrete oriented polytope (k-DOP) [5]. The broadphase binary collision tree is computed on the patch level of granularity using a top-down approach splitting the computed k-DOPs. Next, the broadphase culls patches in parallel over each rank. The goal of this phase is to determine which patches need to be migrated by *ghosting*. Each patch collection element tests collision against a local copy of the broadphase tree. This culling step excludes any patch that is not in the probable-collision step. After a hit in the broadphase tree, a ghosting message is generated for both sides of the collision containing element data for that patch. A ghosting message being received triggers the midphase, where a further culling of element bounding boxes is performed. This cull operates locally by generating a tree using a top-down approach per-element. Then, a narrowphase functor is triggered for each colliding element bounding volume. The final step, narrowphase and enforcement, calculates primitive collision detection and closest point projection of triangular facet collision entities to nodes. Once the gap (relative signed distance) is determined, the penalty force for the contact is computed. The last part of this step is communicating the penalty force back to the original MPI ranks in order to include the contact force in the force assembly (refer to Algorithm 1).

We carried out a preliminary investigation of the performance of our prototype. Although load-balancing is not yet part of the prototype, we hope to understand the basic performance characteristics of the AMT approach. These experiments were performed on two input datasets representing different contact profiles. The first dataset, *sphere/plate contact*, involves a sphere colliding with a wall. This case results in a limited area of contact as only a few elements are in contact. The second dataset, *cubes contact*, exhibits contact between eight cube objects. In this contact scenario, each cube has three faces in contact, meaning one half of surface facets and nodes are in contact. Figure 4 shows the *sphere/plate contact* problem FEM mesh and contact submodel with FE faces decomposed into triangular facets. Note that contact entities are present only on object surfaces, which contributes to the disparity in load balancing

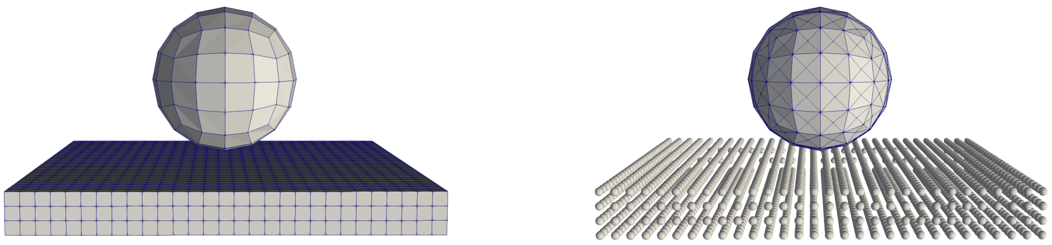

Figure 4: FEM mesh (left) and contact entities (right).

between contact and non-contact routines.

In our experiments, we computed the parallel efficiency of our approach along with timing data for both search and enforcement. Parallel efficiency E indicates the full utilization of parallel cores; it measures the ratio of ideal parallel time (sequential time t_{seq} divided by the number of processors p) to the actual parallel time $t(p)$: $E = \frac{1}{p}\frac{t_{seq}}{t(p)}$. The closer the efficiency E is to 1, the closer the parallel efficiency is to ideal. Our experiments were performed on up to 16 compute nodes of a Dual Socket Intel E5-2683v3 2.00GHz CPU cluster, each node having 28 total processors with 256 GB of DDR3 RAM. We mapped one DARMA/***vt*** rank to each physical core, yielding 28 ranks per compute node.

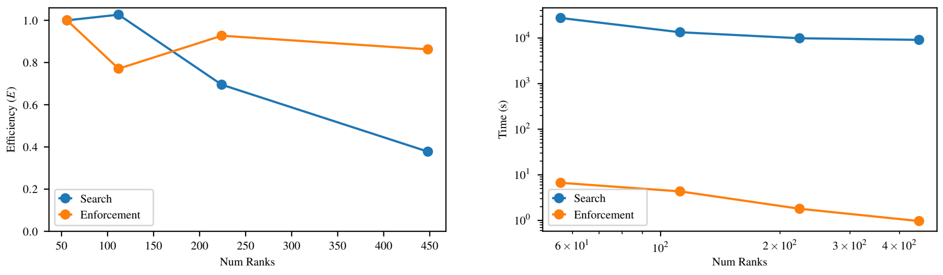

Figure 5: Efficiency plot of our approach on a strong-scaling *sphere/plate contact* problem. Our approach shows that search efficiency stays above around 70% up to about 200 ranks, but dips after that. The timing data clearly shows that search is the most expensive part of our implementation on this problem.

Figure 5 shows the results of our contact scheme on the *sphere/plate contact* problem. Our results show reasonable scaling up to 448 ranks on 28 compute nodes. In our timing experiments, we show a vast difference between the search and enforcement portions of the code due to a large overhead in our search implementation. This indicates that although the tasking model can avoid scaling issues related to singular contact points, the overhead issues associated with the implementation need to be addressed.

Figure 6 shows the results of the DARMA/***vt*** contact scheme on the *cubes contact* problem. These results show that our current implementation struggles for smaller problems. One possible explanation is the relative impact of overhead; the problem uses a total of $130,952$ contact entities, significantly less than the *sphere/plate contact* problem which has $606,546$ entities. With considerably less work to do, overdecomposition becomes more of a hindrance than help, since the overhead cost is per overdecomposed task.

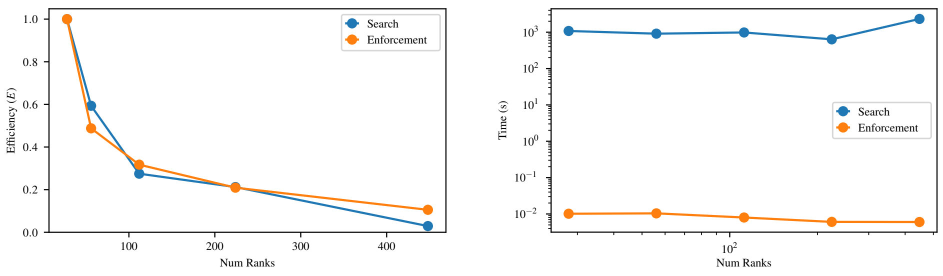

Figure 6: Efficiency plot of our approach on a strong-scaling *cubes contact* problem. On this smaller problem, we see more difficulties with scaling. This is likely because of the overheads initially observed in the *sphere/plate contact* problem.

Future work on the application of DARMA/***vt*** to contact mechanics will focus on the use of AMT to improve load imbalance through the dynamic redistribution of work among available resources and investigating the reduction of task overhead. We are also interested in hybrid *Kokkos*/distributed memory approaches. Work presented in Section 2 describes the advantages to using frameworks such as *Kokkos* for on-node parallelism and the *ArborX* library [6] shows the direct application of *Kokkos* for efficient contact search. It would be possible to use *ArborX* kernels in the midphase/narrowphase portion of our contact search. In the future, a detailed comparison of all three methods, the *ArborX* MPI method, a hybrid method, and the DARMA/***vt*** method would yield interesting insights into the problem of efficient and performance-portable contact.

4 CONCLUSIONS

In this work, we addressed software engineering challenges associated with next-generation computing hardware. We first explored the *Kokkos* package for performance portability. *Kokkos* provides an abstraction layer between the application code and a number of platform-specific backends, allowing developers to write code once and deploy it across disparate architectures. The *NimbleSM* solid mechanics code and LAMÉ constitutive model library were used to demonstrate the effectiveness of *Kokkos* for improving the performance of explicit dynamics simulations on computing platforms equipped with GPUs. A software interface for constitutive models was created to exploit high-performance heterogeneous architectures while maintaining an accessible code design for developer productivity. The use of GPU accelerators via *Kokkos* resulted in performance improvements of $10\times$, or more, relative to CPU-only platforms.

We then investigated the potential of AMT scheduling to improve the performance of contact mechanics algorithms. Contact is particularly amenable to AMT because it evolves dynamically over the course of a simulation and results in severe load imbalance with typical partitioning schemes. The DARMA/***vt*** library was utilized within *NimbleSM* to manage collision detection, contact enforcement, and parallel synchronization of contact forces. DARMA/***vt*** supports asynchronous one-sided communication, virtualization, dynamic load balancing, and interoperability with MPI bulk-synchronous portions of the code. Evaluation of several test problems indicated that DARMA/***vt*** has the potential to improve the scaling of contact mechanics, particularly when contact occurs in isolated regions of the computational domain. Additional work is required, however, to address the overhead expense currently associated with our DARMA/***vt*** contact implementation.

ACKNOWLEDGMENTS

This work was funded by the Advanced Technology Development and Mitigation and Computational Systems and Software Engineering elements of the U.S. Department of Energy's Advanced Simulation and Computing program. Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government. SAND2021-2224 C.

REFERENCES

[1] J.C. Bennett, M.T. Bettencourt, R.L. Clay, H.C. Edwards, M.W. Glass, D.S. Hollman, H. Kolla, J.J. Lifflander, D.J. Littlewood, A.H. Markosyan, S.G. Moore, S.L. Olivier, J.A. Perez, E.T. Phipps, F. Rizzi, N.L. Slattengren, D. Sunderland, and J.J. Wilke. ASC ATDM level 2 milestone #6015: Asynchronous many-task software stack demonstration. Report SAND2017-9980, Sandia National Laboratories, Albuquerque, NM and Livermore, CA, 2017.

[2] H.C. Edwards, C.R. Trott, and D. Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12), 2014.

[3] D. Harmon, E. Vouga, B. Smith, R. Tamstorf, and E. Grinspun. Asynchronous contact mechanics. In *SIGGRAPH 2009 papers*. Association for Computing Machinery, 2009.

[4] L.V. Kale and S. Krishnan. Charm++: A portable concurrent object oriented system based on C++. In *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '93. Association for Computing Machinery, 1993.

[5] J.T. Klosowski, M. Held, J.S.B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Transactions on Visualization and Computer Graphics*, 4(1), 1998.

[6] D. Lebrun-Grandié, A. Prokopenko, B. Turcksin, and S.R. Slattery. ArborX: A performance portable geometric search library. *ACM Transactions on Mathematical Software*, 47(1), 2020.

[7] J.J. Lifflander, P. Miller, N.L. Slattengren, N. Morales, P. Stickney, and P.P. Pébaÿ. Design and implementation techniques for an MPI-oriented AMT runtime. In *2020 Workshop on Exascale MPI (ExaMPI)*. IEEE, 2020.

[8] D.J. Littlewood and M.R. Tupek. Adapting material models for improved performance on next-generation hardware. Memorandum SAND2017-5873, Sandia National Laboratories, Albuquerque, NM and Livermore, CA, 2017.

[9] X. Ni, L.V. Kale, and R. Tamstorf. Scalable asynchronous contact mechanics using Charm++. In *2015 IEEE International Parallel and Distributed Processing Symposium*, 2015.

[10] W.M. Scherzinger and D.C. Hammerand. Library of Advanced Materials for Engineering – LAME. SAND Report 2007-5515, Sandia National Laboratories, Albuquerque, NM and Livermore, CA, 2007.

[11] SIERRA Solid Mechanics Team. Sierra/SolidMechanics 4.58 User's Guide. SAND Report 2020-10045, Sandia National Laboratories, Albuquerque, NM and Livermore, CA, 2020.