Engineering with Computers manuscript No.
(will be inserted by the editor)

An MPI+*X* Implementation of Contact Global Search Using Kokkos

Glen A. Hansen[a] · **Patrick G. Xavier**[b] · **Sam P. Mish**[b] · **Thomas E. Voth**[a] · **Martin W. Heinstein**[c] · **Micheal W. Glass**[d]

Received: date / Accepted: date

Abstract This paper describes an approach that seeks to parallelize the spatial search associated with computational contact mechanics. In contact mechanics, the purpose of the spatial search is to find "nearest neighbors," which is the prelude to an imprinting search that resolves the interactions between the external surfaces of contacting bodies. In particular, we are interested in the *contact global search* portion of the spatial search associated with this operation on domain-decomposition-based meshes. Specifically, we describe an implementation that combines standard domain-decomposition-based MPI-parallel spatial search with thread-level parallelism (MPI-X) available on advanced computer architectures (those with GPU coprocessors). Our goal is to demonstrate the efficacy of the MPI-X paradigm in the overall contact search.

Standard MPI-parallel implementations typically use a domain decomposition of the external surfaces of bodies within the domain in an attempt to efficiently distribute computational work. This decomposition may or may not be the same as the volume decomposition associated with the host physics. The parallel contact global search phase is then employed to find and distribute surface entities (nodes and faces) that are needed to compute contact constraints between entities owned by different MPI ranks without further inter-rank communication. Key steps of the contact global search include computing bounding boxes, building surface entity (node and face) search trees and finding and distributing entities required to complete on-rank (local) spatial searches.

To enable source-code portability and performance across a variety of different computer architectures, we implemented the algorithm using the Kokkos hardware abstraction library. While we targeted development towards machines with a GPU accelerator per MPI rank, we also report performance results for OpenMP with a conventional multi-core compute node per rank.

Results here demonstrate a 47% decrease in the time spent within the global search algorithm, comparing the reference ACME algorithm with the GPU implementation, on an 18M face problem using 4 MPI ranks. While further work remains to maximize performance on the GPU, this result illustrates the potential of the proposed implementation.

Keywords Partial differential equations · finite element analysis · contact problems · spatial searching

Mathematics Subject Classification (2000) 65Y05 · 65Y25 · 68P10 · 74M15

[a] Computational Multiphysics Dept. 1443

[b] Simulation Modeling Sciences Dept. 1543

[c] Computational Solid Mechanics & Structural Dynamics Dept. 1542

[d] Computational Simulation Infrastructure Dept. 1545

Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1321, USA
[a]E-mail: gahanse@sandia.gov

1 Introduction

1.1 Overview

Treatment of contacting material surfaces within a mechanics simulation is one of the more time consuming activities in a typical engineering calculation. Within such a calculation, one often needs to analyze the behavior of multiple deforming bodies that are moving with respect to each other, such as can occur when modeling fragment witness-plate interactions, as well as problems where only a single, self-contacting surface is of interest.

Examples of contact include the thermomechanical behavior of nuclear reactor fuel coming into contact with its protective cladding [1], vehicle crash simulation, tire performance simulation, projectile calculations, chip behavior during machining, and many others. In most of these applications, calculations of the underlying physics is expensive and the addition of contact significantly increases this expense. Thus, the development of efficient computing strategies for contact problems is quite important. The evolution of computer architectures is driving a need to develop multithreaded and coprocessor-based configurations that work within MPI-based applications to provide higher degrees of parallelism to support ever more complex analyses.

Our goal is to study the combination of MPI-level and thread-based parallelism within an existing contact library, that includes correctness and performance tests to allow us to ensure that regression in the overall capability does not occur as new algorithms are introduced. We directly focus on the parallel, domain-decomposed, contact global search. In this paper, we define *contact global search* as the parallel portion of the spatial search phase of contact detection. Spatial search (also called the proximity search) provides a list of entities in the neighborhood of another entity. The imprinting (fine) search associated with contact detection, where the actual interactions between entities is determined, is not considered here. We hypothesize that the algorithms developed here may be useful in supporting the imprinting search, however.

Section 1.2 motivates and describes this study. To be representative and to provide near term impact, we took the approach of modifying an existing contact library that is used in analysis codes on massively parallel architectures. The ACME library [2], described in Section 2, was selected due to its availability and familiarity with its use inside Sandia. Profiling the current behavior of ACME on a test problem (Section 2.1) was used to assess the run-time significance of contact global search within a typical analysis application.

In Section 3 we examine ACME's contact global search algorithm for on-rank, thread-based parallelism opportunities. We feel that ACME provides a realistic framework for this discovery process. Our experiences suggest that other contact libraries support traditional parallelism in a similar fashion to ACME; thus we expect that the approach studied here would be transferable to those libraries. We implement an MPI plus on-rank threading algorithm (MPI+X) that exploits on-rank parallelism within the contact global search. To enable code portability and consider performance portability between machine architectures, we use the Kokkos manycore abstraction library in our implementation. Source-level and compile-time options are used to select among different low level implementations of on-node parallelism, such as CUDA when a GPU is available and OpenMP for conventional multicore processor nodes and the Xeon Phi. Although we primarily targeted machines supporting CUDA/GPUs, we report performance results that also include for OpenMP with a conventional multicore compute node per rank (Section 4).

1.2 Background and General Approach

The need to accurately predict the location and evolution of contacts between multiple surfaces undergoing large relative motions can be critical to a calculation. Consider a billiards problem; the transient relationship between two balls, the location of the contact point when they touch, and the velocity (and resultant force) that they impart on each other completely govern the outcome of the interaction. Accuracy of the result may be the difference between simulating a table clearing shot versus no balls sunk. Add the complexity of large deformations and self-contact (i.e. a surface contacting itself), and it is clear that simulation of contact mechanics can be an expensive proposition.

A contact simulation typically begins with identifying the discrete entities that make up the external surfaces of the bodies being simulated, to determine the sets of entities that could potentially contact each other. In many cases, particularly in the case of a finite element simulation, the bodies are meshed using volume elements. The surfaces of these bodies are then represented by the external faces and nodes of the volume mesh.

The contact search operation amounts to determining which of the external discrete entities (nodes, element edges, and faces) of one body might come into contact with the entities of the second body, during the current time step[1] of the simulation code. This determination is not typically well posed in the sense that the position of the entities on each of the bodies is known at the beginning of the time step, but they might not be known at the end of the time step. For the purposes of this paper, we will assume that the entities move in a simple way during a time step and traverse the next highest dimension in space; *i.e.,* a node traces out a curve as it moves in time, an edge traces out a quadrilateral region, and a face sweeps out a volume as it advances during a time step.

The goal of the contact search is then to determine which of the swept areas and volumes intersect within a given simulation time step. To perform this intersection search efficiently a spatial search is first performed. In this step one will often place these swept regions into ε-inflated, axis-aligned bounding boxes (*c.f.* Figure 1), and then employ a binary or tree search to determine which of the bounding boxes potentially intersect. The use of a k-D tree search

[1] Most of this discussion regarding the search operation is independent of whether the simulation is transient or quasi-static. We choose to use the term *time step* rather than *load step* here, but they may be used interchangeably unless otherwise noted.

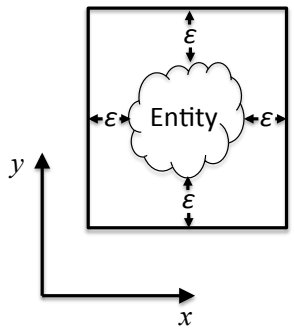

Fig. 1 An ε-inflated axis-aligned bounding box is aligned with the x and y axes, and encloses the entity in question with at least an ε separation between the outside surface of the entity and the inside surfaces of the bounding box.

scheme is a popular spatial searching approach (*c.f.* [3]). Attaway *et al,* [4] present more details on the construction of bounding boxes for contact search in dynamics applications, where the potentially contacting entities move within a time step. As noted earlier the imprinting search which finally resolves the precise contact pairs is not discussed here.

Typically, the number of bounding boxes N that need to be searched each time step can be quite large, as they correspond to the entities that lie on the external boundaries of the meshed bodies. Given an efficient tree search, one can typically determine the intersection candidates in $O(N\log(N))$ operations, where $\log(N)$ operations are needed to scan the tree for each item in the set of N items participating. One usually desires to perform this tree search operation in parallel to reduce run time. Secondly, the underlying computational mechanics simulation is also run in parallel, often using a domain decomposition method to distribute the finite elements evenly on the available processor ranks (the *primary decomposition*) in an MPI-parallel system. The surface entities will inherit the underlying finite element decomposition structure, but this typically does not result in a properly load-balanced contact spatial search decomposition. Indeed, for an arbitrary finite element decomposition it is likely that some subset of the active processors will possess a considerable number of boundary entities that need to participate in the spatial search, and some ranks will have none. The parallel spatial search using such a decomposition will not perform well due to this imbalance and the fact that not all of the allocated processors are used for the contact operations.

There are two potential approaches to parallelizing the spatial search at the MPI level:

- Re-partition the surface entities to obtain a balanced *secondary decomposition* and perform the spatial search using that decomposition, or
- Use the primary decomposition for the on-rank spatial search and "ghost" potential contact entities from "nearby" processor ranks and just accepting a degree of load imbalance during this local search operation. Note that this imbalance can be reduced somewhat by using a spatial decomposition (*e.g.,* recursive coordinate bisection, RCB) rather than a graph based decomposition for the finite element mesh partitioning.

Clearly, the decision to use a primary vs. secondary decomposition approach amounts to considering the tradeoff between parallel efficiency and entity communications, and is problem and decomposition dependent. The secondary decomposition approach, while well balanced for the contact operations, requires a large amount of inter-rank data communication each time step if the primary and secondary decompositions are significantly different.

In the primary decomposition approach, the load balance associated with the on-rank spatial search is generally not ideal (as it is driven by the volume decomposition). Here, ghosting is used to copy entities that are spatially close to a given processor's entity set to that processor, such that no further inter-processor communications are needed to complete the on-rank search operation. While the load balance may be poor, generally much less inter-processor communication is needed relative to the secondary decomposition approach.

The "best" method to use is typically problem and hardware dependent, as one desires to minimize the "wall clock" time spent in the search operation. The search includes the time needed for the parallel search plus the communications time needed to set up and recover from the search. One can envision problems where a secondary decomposition is ideal from an overall performance standpoint, and one can also envision the opposite scenario where ghosting would be superior.

The goal of this work is to demonstrate the extension of a standard parallel contact search execution model to use manycore processing within an MPI rank. In order to explore on-rank parallelism, our desire was to focus on a mature, representative, MPI-parallel contact library with an existing set of regression (both correctness and performance) test problems. The Algorithms for Contact in a Multiphysics Environment (ACME) library [2] meets those criteria. ACME implements entity migration between processors, domain decomposition, and load balancing using the Trilinos Zoltan communications library [5].

We concentrated on the contact global search function as this is often the most time consuming aspect of an analysis employing contact. Further, we consider the primary decomposition approach outlined above and do not address the use of the secondary decomposition technique. To further the effort, we profiled the behavior of ACME within the ALEGRA shock hydrodynamics application on a benchmark problem; see Section 2.1. The profiling results revealed that much of the work in contact global search is

spent calculating the entities and data that must be ghosted from one MPI rank to another. Thus, we focused on accelerating the ghosting search operations performed within an MPI rank in ACME. Our strategy is based on the philosophy that most of the work within a rank can be performed on a coprocessor; given the degree of parallelism that might be available in the future, a significant degree of parallel load imbalance between any two MPI ranks may become a secondary consideration. Stated another way, if floating point operations within a rank are very efficient (the "FLOPS are free" philosophy), MPI parallelism becomes a less impactful source of performance.

This presentation begins with Section 2, which describes the details of the MPI-based contact global search implementation in ACME that serves as the reference implementation. Section 3 summarizes the structure of the ghosting search and how we chose to structure it for GPU execution. Significant restructuring was performed to improve the data movement between the ("host") processor and GPU coprocessor. We modified the ghosting search approach to support efficient calculations of intersections between two sets of axis-aligned bounding boxes (AABBs), as this operation is performed three times in the overall approach. Many contact search algorithms are based on the use of a k-D tree. Unfortunately, k-D tree construction can be problematic on on GPU architectures as insufficient parallelism exists at the top of the tree [6]. Thus, an algorithm specifically designed for spatial searching on GPUs uesd here. We chose Karras's Morton code linearized bounding volume hierarchy (BVH) construction algorithm and parallel AABB BVH overlap search method [6,7]. Finally, in order to provide sufficient parallelism in the remainder of the on-rank work, we substantially restructured the last part of the ghosting algorithm where the AABB intersection operations are performed.

1.3 Kokkos for Portability

There are multiple models that support multiple threads of execution within a single MPI rank. The combination of MPI with such a model referred to as an MPI+X approach. Example "+X" software environments include OpenMP, Pthreads, CUDA, OpenCL, CilkPlus, Threading Building Blocks, and Microsoft's Task Parallel Library [8–14]. Examples of hardware that supports these environments include conventional multicore CPU chips, Intel's Xeon Phi, and GPUs or FPGAs used as coprocessors. CUDA is an example of a programming model that is closely tied to GPU coprocessor hardware models.

Sandia's Kokkos library [15–17] provides a programming abstraction layer intended to enable developers to write source code that is portable across multiple execution environments. It is implemented as a layered collection of templated C++ libraries that provide a thread-parallel programming model. Compile-time options select the specific underlying programming model (CUDA, OpenMP, or PThreads) that is used in the generated code. Further, one desire is that Kokkos provide *performance portability*; it is designed to give a significant fraction of the performance that would be obtained if the code for a given algorithm was written specifically for the architecture in question [17].

Kokkos includes *parallel_for*, *parallel_reduce*, and *parallel_scan* control structures, as presented in Section 3.1. The semantics of these abstractions in Kokkos are similar to other libraries [13,10,18], and thus they will not be defined here. Several additional features of Kokkos address code portability. To support CUDA, Kokkos explicitly includes the concept of multiple memory and execution spaces. Template specialization is used to generate C++ code for the desired execution model. For example, the *Kokkos::View* class provides an abstraction for multi-dimensional arrays that includes a template parameter that specifies the memory space where the data resides (regular "host" memory or in the GPU's memory space). An additional template parameter controls data layout, which is used to optimize memory accesses inside interior loops contained within the body of a parallel loop (*e.g.,* a *parallel_for*). The *Kokkos::DualView* class can be used to manage a pair of *Views* with matching data layouts, providing a convenient mechanism for synchronization of data between different memory spaces.

The target for this study was a MPI+GPU environment. However, using Kokkos we were able to create a single-source implementation that produced executables for OpenMP, PThreads, or CUDA. This allowed us to pursue development and debugging of thread-safe implementations of our approach on conventional multi-core systems. Debugging the alternate implementations focused on issues with copying information between memory spaces. Initial development was performed on a MPI+GPU platform, a Cray XK7 with NVIDIA K20x GPUs using one GPU per MPI rank. Performance data was obtained on a partition of Sandia's Shannon testbed with NVIDIA K80 GPUs, using one Sandy Bridge core and one GPU per MPI rank.

Section 3 describes the modifications to the contact global search algorithm that were performed. Various parallel programming idioms were applied to support the incorporation of the parallel BVH-construction algorithm developed by Karras. We restructured the ghosting algorithm to increase parallel performance. The use of Kokkos supports a comparison of the performance of the ghosting search algorithm between GPU and conventional multicore architectures. Section 4 presents this data, using the OpenMP model for the multicore case. We conclude with suggestions on how additional performance improvements might be achieved, and other directions for future work.

2 Ghosting for Contact Global Search

We chose the existing ACME contact library to serve as a reference for this work. In this section, we describe the profiling performed on ACME to provide a performance baseline, and to focus efforts on the highest priority sections of the code to address. Finally, we summarize the ACME contact global search problem and the redesign activities we pursued.

2.1 Contact Global Search in ACME

Algorithms for Contact in a Multiphysics Environment (ACME) is a mature, widely used multibody contact library developed at Sandia National Laboratories [2]. The library implements various algorithms to i) search for potential interactions between body surfaces represented by analytic and discretized topological entities and ii) determine interaction forces needed to prevent penetration or other violation of surface integrity. The typical application of ACME is to support other software that simulates bodies in contact by forming and integrating equations of motion. ACME is written in terms of geometry and possible interactions between discrete surface entities and representations (nodes on the surface of a body and faces that connect nodes to represent the body's surface). The ACME code base amounts to approximately 90,000 lines of C++ code contained in include, implementation, and driver files.

ACME contains a set of 89 correctness tests, of which roughly half are designed to execute in parallel using MPI on four processors. During this study, the "correctness" of the evolving code base was verified by running these tests prior to committing changes to the source repository, to ensure that code modifications do not alter the basic features, reliability, and capabilities of the library as it was modified.

We developed a set of six performance tests to guide the development process. These tests are based on computing the time required to calculate the contact constraint system between bricks in a wall (Figure 2). The performance test problems range from 16K to 18M surface elements (faces). Note that the mesh needed for the full physics simulation includes many more volume elements but this number is not relevant to the contact search and so we chose to report results in terms of the number of element faces that define the contact surface.

We selected the 4458K surface face problem as a point of discussion for many of the profiling results, as it is of large enough size to give good repeatability in the timing calculations. We focused on using a primary decomposition obtained by the Zoltan [5] recursive coordinate bisection (RCB) algorithm. This is a good performance strategy for the "brick wall" test problems as RCB gives an even decomposition across brick boundaries, where a graph-based algorithm would not as the mesh connectivity graph of each brick is isolated from its neighbors.

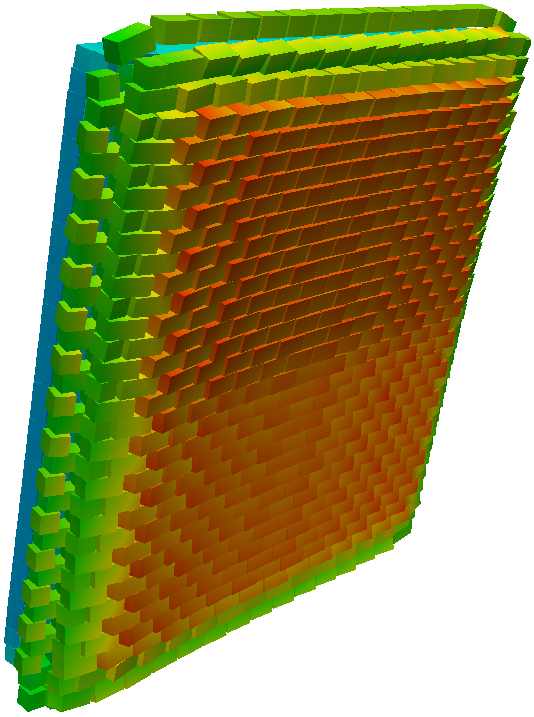

Fig. 2 "Brick Wall" performance test problem geometry. Here, the ALEGRA code is used to simulate a pressure pulse in front of the wall that imparts a force on the bricks and displaces them as seen in the diagram.

To help understand the distribution of work in the setup of ACME and the layout of the work performed each time step, Sandia's ALEGRA [19] code was used together with `gprof` [20] to obtain a profile of a dynamics calculation involving the brick wall problem. The ALEGRA code uses ACME for contact detection and enforcement. This profiling result was used to isolate the sections of ACME that involve functions executed more than once during a given simulation (*i.e.,* those operations performed each time step of the simulation). The ACME driver was then used to reproduce the decomposition used in this ALEGRA problem, up to the point of assembling the contact constraints into the host code's finite element data structures.

A key excerpt from the profiling graph is shown in Figure 3. The most time-consuming section of the code was the contact search module of the code, which encompasses both the contact global search and the imprinting search. This module was responsible for 34% of the overall run time of the ALEGRA simulation. Within that module, the most expensive operation was the contact global search, which calculates the face ghosting operations needed to support the node-face contact model. Given that 18% of the time is spent within this ghosting function, we first

focused on it for the conversion to Kokkos coprocessor execution. The specific function that implements ghosting is named `ContactSearch::DoGhosting_New_NodeFace`; we will refer to it as the *contact ghosting function* throughout the sequel.

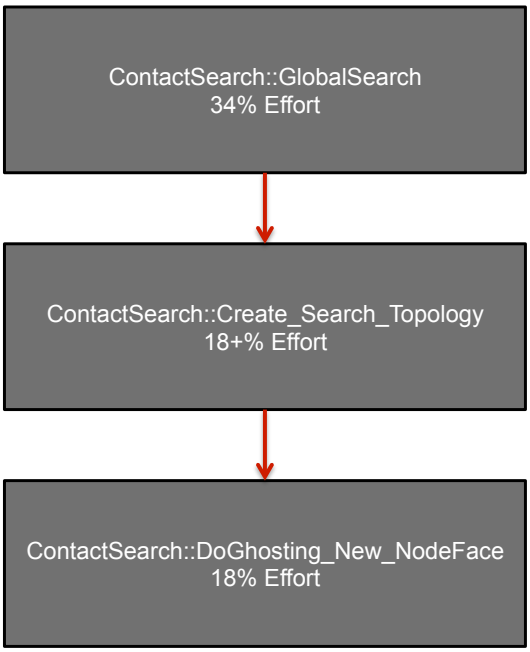

Fig. 3 Overall effort spent in the immediate children methods of `ContactSearch::GlobalSearch`, as a percentage of total execution time of the ACME driver.

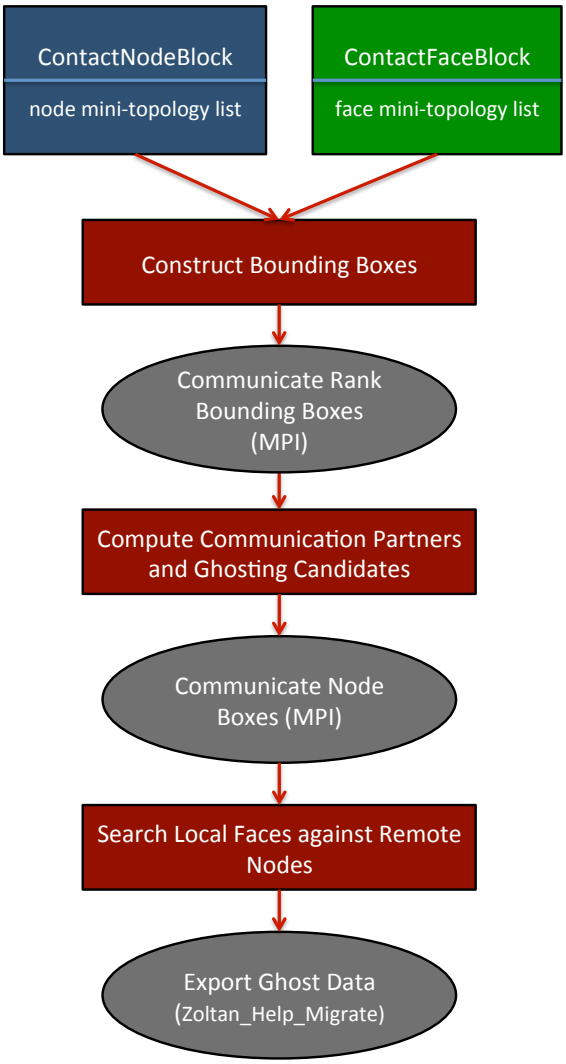

Fig. 4 Functional layout of the operations performed inside the contact ghosting function, `ContactSearch::DoGhosting_New_NodeFace`.

2.2 Ghosting for Contact Global Search

We now describe the algorithm used by ACME for contact global search, with emphasis on determining which entities to ghost. Figure 4 shows the structure of the ACME approach. At the highest level, it uses a global search to locate off-rank nodes and faces that are needed to support an on-rank (local) search. It is this specific algorithm that we adapt in later sections to exploit on-rank parallelism.

The first phase of the algorithm is to *construct bounding boxes* on each rank from the lists of the nodes and faces on that rank. Specifically, it creates a list of ε-inflated axis aligned bounding boxes (AABBs) for each of these entities. Let b_n denote an ε-inflated axis aligned bounding box for a node, and b_n^k is the box for the k^{th} node on the rank. The set of bounding boxes for all the nodes k on a given rank is,

$$B_n = \{b_n^1, b_n^2, \ldots, b_n^K\}. \tag{1}$$

Furthermore, each rank constructs two additional ε-inflated bounding boxes, each large enough to contain all the nodes and faces on the rank, respectively. For brevity, we sometimes refer to these AABBs as the *rank bounding boxes*. The all-nodes bounding box (that is the rank bounding box for all nodes on a given rank) is

$$\overline{B_n} = bbox(\cup b_n^k). \tag{2}$$

For the faces on the rank, the ε-inflated bounding boxes for the faces are denoted,

$$B_f = \{b_f^1, b_f^2, \ldots, b_f^K\}, \tag{3}$$

and the rank's all-faces bounding box is,

$$\overline{B_f} = bbox(\cup b_f^k). \tag{4}$$

The next phase in the ghosting algorithm is to *communicate rank bounding boxes*. This is an all-to-all (MPI) communication of each rank's all-nodes and all-faces bounding boxes $\overline{B_n}$ and $\overline{B_f}$. The result of this operation is that each

rank knows the physical extents of all of the other ranks' all-node and all-face bounding boxes.

Each rank now has the information needed to independently and consistently compute its communication partners for the rest of the ghosting algorithm. Specifically, in the *compute communication partners and ghosting candidates* phase, each rank r will determine

i) the set of ranks Q_{rcv} from where r will receive a set of node bounding boxes,
ii) the set of ranks Q_{snd} to where r will send a set of node bounding boxes,
iii) and for each rank $q \in Q_{\text{rcv}}$, the set of local face bounding boxes for which it will need to check for overlaps against the node bounding boxes to be received by r from rank q

as outlined in the subsequent text.

For the remainder of the ghosting algorithm, we expect each rank to need to exchange data with relatively few other ranks. Until the final phase, the data exchanged will be lists of bounding boxes. Knowledge of the communication partners is important as such information will allow pairwise MPI communication rather than requiring a MPI_Alltoallv operation.

The first step of the *compute communication partners and ghosting candidates* phase is for each rank r to create a bounding volume hierarchy (BVH), or search tree, of the all-nodes bounding boxes $\{\overline{B_n^q}\}$ and a BVH of the all-faces bounding boxes $\{\overline{B_f^q}\}$ from the other ranks $q \neq r$. If $\overline{B_f^r}$ overlaps $\overline{B_n^q}$ then $q \in Q_{\text{rcv}}^r$; this completes step (i). If $\overline{B_n^r}$ overlaps $\overline{B_f^q}$ then $q \in Q_{\text{snd}}^r$; which completes step(ii).

Each rank then uses the BVH of $\{\overline{B_n^q}\}$ against its own local face boxes B_f to determine which of the b_f^k overlap which $\overline{B_n^q}$ for the remote rank q. Similarly, each rank does the same for B_n and the BVH of $\{\overline{B_f^q}\}$: find which b_n^k overlap which $\overline{B_f^q}$. The node boxes b_n^k on the local rank that overlap the remote rank q's all-faces bounding box $\overline{B_f}$ are added to the node box ghosting list L_{gns}^q for that rank q, as each such node is a candidate to interact with a face that might be ghosted over from rank q. Similarly, the overlapping face boxes b_f^k on the rank are stored in its face box ghost list L_{gfs}^q. The faces whose boxes are in L_{gfs}^q are candidates to be ghosted over to rank q.

The *communicate node boxes* phase involves an MPI send operation, where the members of the node box ghosting lists L_{gns}^q created above are sent to the respective remote ranks $q \in Q_{\text{snd}}$. The incoming node boxes from a rank $q \in Q_{\text{rcv}}$ are placed in the received node box ghost list L_{gnr}^q.

The final computational phase is to *search local faces against remote nodes*. Here, each rank loops over each remote rank q and compares the list of node boxes L_{gnr}^q received from q with the face box ghost list L_{gfs}^q. For each non-empty L_{gnr}^q, a BVH of L_{gnr}^q is computed and used to accelerate this overlap search. If an incoming node box in L_{gnr}^q is found to overlap with a face box in L_{gfs}^q, then the face that spawned the face box is marked for ghosting and is added to a ghosted entity list L_{ges}^q, to be sent via to the destination rank q.

Once all the ranks complete the construction of their L_{ges} lists, the *export and migrate ghost data* phase completes the ghosting by performing the data marshaling, data movement between ranks, and data de-marshaling.

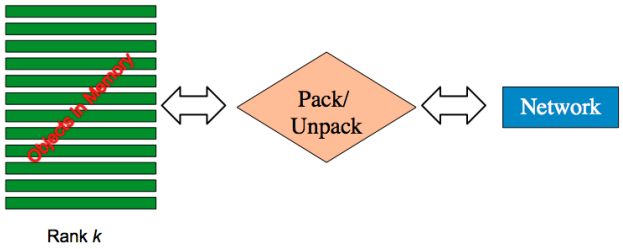

Fig. 5 Data flow between main memory and the communications network on a conventional MPI-parallel system.

As the MPI+X model can include multiple memory spaces on each MPI rank, we want to first review the MPI-based communication phases in the ghosting algorithm. The first of these is an all-to-all exchange of each rank's all-nodes and all-faces bounding boxes. In the second, each rank sends L_{gns}^q to rank q for each q for which any $b_n^k \in B_n$ overlaps $\overline{B_f^q}$. The third is the actual ghosting operation, where the on rank faces boxed by the $\{b_f^k\}$ that overlap with a ghosted node box received from rank q are copied to rank q, in effect "ghosting" these faces on rank q. Figure 5 illustrates the data flow during these communication operations. Here, the entity bounding boxes (nodes and/or faces) exist as objects in main memory. These objects are assembled into communication buffers, and sent over the inter-processor communications network using MPI. Once received by the target processor, the buffers are unpacked to form ghosted entities in the main memory of the target rank.

Within ACME, significant optimization of the above search and MPI communications path has occurred over years of use and performance optimization activities. As we proceed to develop new approaches for the search and data communications model by moving to a multicore implementation, all performance comparisons will be performed against the optimized ACME implementation.

3 Manycore Parallelization of Contact Global Search

We now discuss our development of the manycore-ready approach to transform the MPI+serial contact ghosting function just described. Again, the primary decomposition ap-

proach is used with entities ghosted to each processor so that the local search may be performed without further interprocessor communication.

3.1 Parallel Programming Idioms

We first review several programming idioms that are useful for implemented threaded parallel algorithms. We will employ these terms in our description of on-rank parallelization of the contact ghosting function.

The concepts of *parallel for*, *parallel reduce*, and *parallel scan* [21–24, 13] are becoming increasingly common in parallel programming practice, as they are featured in a number of threading libraries and extension languages [18, 13, 16] [2]. The description here most closely follows the terminology used in the literature and documentation on Kokkos [16, 25] and Threading Building Blocks [13, 26].

The *parallel_for* construct is similar to a traditional *for* statement

for (int iw = 0; iw < worksize; iw++) {
/ Loop Body */ }*

but additionally specifies that the increments of work done by executions of the loop body should be performed asynchronously without regard to any dependencies. The loop body can be encapsulated by a function object, or *functor*, whose inline member function [3]

void operator() (int iw) const { / work */ }*

performs the work increment *iw* when called with *iw* as its argument. [4] The variables used in the loop body but declared outside of it need to be bound to member variables of the functor in its declaration.

parallel_for(worksize, functor_inst);

then applies a functor instance, *functor_inst*, in a thread parallel fashion.

parallel_reduce incorporates *parallel_for* semantics but additionally enables computation of a summary result from results computed by the individual applications of the functor. This summarizing computation is known as a *reduction* with the summary result variable called the *reduction variable*. A binary *reduction operator* is used to combine results, and the reduction variable is first initialized to the identity value (zero for addition and one for multiplication). Conceptually, when expressed in serial terms, the reduction has the form:

for (int iw = 0; iw < worksize; iw++) {
$var_{red} = op_{red}(var_{red},\ result_{iw})$*; }*

where var_{red} is the reduction variable and $result_{iw}$ is the result of some operation associated with work increment *iw*.

For *parallel reduce* the reduction operator op_{red} must be both associative and commutative—for example: addition, max, or min. In both TBB and Kokkos, the functor provides *init(.)* and *join(..)* member functions for the initialization and reduction operation on the reduction variable. The functor's operator has the signature

void operator() (int i, value_type &val_{red}*) const;*

The *parallel_reduce* implementation calls this operator to get each work increment's contribution to the reduction value and uses *join(..)* to perform the reductions.

parallel_reduce(worksize, functor_inst, var_{red}*);*

invokes the functor member functions to perform the work asynchronously and carry out a parallel reduction algorithm.

parallel prefix scan, or *parallel scan*, is a generalization of prefix sum. The inclusive prefix sum of a sequence of numbers $\{a_i\}$ is the sequence $\{\sum_{j=0}^{i} a_j\}$. The exclusive prefix sum excludes the $j = i$ case from each summation. Conceptually *prefix scan* generalizes from numbers to arbitrary data type and to any associative binary operation on that data type. The Kokkos *parallel_scan* API is similar in structure to the *parallel_for* and *parallel_reduce* API. An application code functor is responsible for providing member functions that the *parallel_scan* implementation calls to execute the "loop body" work and carry out the parallel prefix scan algorithm.

Program correctness with respect to multiple execution threads reading and writing shared data is termed *thread safety*. The two main idioms for enabling thread-safety with respect to shared modifiable data [5] are mutual exclusion, also known as "locking", and atomic operations. While locking is general, it can entail unnecessary overhead, and it can complicate multi-threaded software, especially when situations when multiple locks need to be held at once. Atomic primitive functions in a programming model can simplify the implementation of thread-safe algorithms. A function is atomic if the effect of concurrent threads executing it can always be produced by some serial execution.

[2] While OpenMP does support *parallel reduce* by providing a *reduction* construct that can be combined with *parallel for*, it does not provide similar support for *parallel scan*. An implementation of a parallel prefix scan algorithm for OpenMP must be provided externally.

[3] Note that in the Kokkos API the functor's work increment is given by an integer instead of a range to permit *parallel_for* to compile into a kernel launch on a GPU.

[4] Both Kokkos and TBB APIs for *parallel for*, *parallel reduce*, and *parallel scan* support the use of C++11 lambda objects in place of functors. In certain situations, this use of lambda objects can improve the readability of code. We find the functor-based APIs are easier to describe at the current stage of C++11 feature adoption, and in their full form they are more general.

[5] As distinguished from data that once written is constant.

3.2 Parallelism available within the search

The computational phases of the ghosting algorithm described in Section 2.2 offer several opportunities to exploit manycore parallelism. Except when MPI is explicitly mentioned, our discussion of parallel computation here will refer to work done on a single, arbitrary rank.

In the *construct bounding boxes* operation, the bounding boxes of the individual nodes (and faces) on a rank can be computed independently of each other. Therefore, each of these two sets of boxes, B_n and B_f (see Section 2.2) can be computed in parallel. Further, computing the rank bounding boxes $\overline{B_n}$ and $\overline{B_f}$ can be performed in parallel.

We developed functors for computing the node and face bounding boxes. The functors' constructors are used to provide them with read-only *Views* of the node and face IDs, entity type information, connectivity data, and position information, and writable *Views* to hold the output AABB data.[6] The functors are used through the *parallel_reduce* construct described earlier. We defined a bounding-box structure for both $\overline{B_n}$ and $\overline{B_f}$, and this is used for the reduction variable in both cases. The functors share a common relative base class that provides an implementation of the reduction operation and the initialization for this bounding-box structure.

The parallel construction of the set of face bounding boxes B_f and the of the all-faces bounding box $\overline{B_f}$ is the easier case. The *FaceComputeBoundingBoxFunctor*'s operator uses an work increment argument to index into a *View* of face IDs and writes the corresponding face ε-inflated axis aligned bounding box data into its AABB *View* representation. It also expands the geometric extent of the second argument as needed to contain the inflated AABB.

Parallel construction of B_n and $\overline{B_n}$ is slightly less straightforward because not all the nodes in the input to the functor are used. Specifically, for ACME "boundary" nodes are shared across multiple processors with only one processor rank actually "owning" the shared node. ACME packs the bounding boxes into an array (presumably to save memory), and we chose not to modify this behavior. The *NodeComputeBoundingBoxFunctor*'s operator evaluates a predicate to decide whether the node specified via its index argument will be included. If not, then the node's ε-inflated AABB is not computed, and thus it is excluded from the output *View* of AABBs and from reduction into $\overline{B_f}$. If the predicate evaluates to *false* (the node is not owned by this rank) the operator can simply return. However, if it evaluates to true, the functor must determine the index at which the node's AABB data should be written in the output AABBs *View*.

We tried the simplest approach, using *atomic fetch-and-add* to provide each outputting thread with the next available unused index.

[6] Node and face IDs are simply indexes into data arrays.

idx = Kokkos::atomic_fetch_add(&m_idx(), 1);

Here, *m_idx* is the atomic variable[7] holding the index at which the unoccupied portion of the result *View* data begins. The atomic operation increments the variable at the data location *&m_idx()* and returns the value of the variable before it was incremented. This approach provided good performance on the GPU.

Other opportunities for parallelism arise in both the *compute communication partners and ghosting candidates* and the *search local faces against remote nodes* operations. Recalling Section 2.2, we observe that on each rank the ghosting algorithm performs the following bounding box intersection searches:

i) Between members of $\{\overline{B_f^q}\}$ and $\overline{B_n}$, providing $\{\overline{B_f^{q\cap}}\}$ for a result
ii) Between members of $\{\overline{B_n^q}\}$ and $\overline{B_f}$, calculating $\{\overline{B_n^{q\cap}}\}$
iii) Between members of B_n and members of $\{\overline{B_f^{q\cap}}\}$
iv) Between members of B_f and members of $\{\overline{B_n^{q\cap}}\}$.
v) For each of the other ranks q, between members of L^q_{gfs} and L^q_{gnr}.

Parallel approaches to search for spatial overlaps between two sets of AABBs $\{B_\alpha\}$ and $\{B_\beta\}$ once there is a BVH of either set are well understood. In serial, it is common to implement a single AABB's recursive traversal of a BVH to search for overlaps with its leaves by using a stack, instead of using a search function that calls itself recursively. Let us assume that B_β has fewer members than B_α, and that the BVH for $\{B_\beta\}$ is well-balanced[8]. Then, the outer loop over $\{B_\alpha\}$ can be converted to a thread-parallel implementation with each active traversal having its own array to use as a stack. For each member $b_i^\alpha \in B_\alpha$, it is easy to implement an overlap search recursively searches the BVH of B_β to find all the AABBs $b_j^\beta \in B_\beta$ that b_i^α overlaps.

Some attention is required in storing the results of the search. The basic idea is to use *atomic fetch-and-add* similarly to what is done in *NodeComputeBoundingBoxFunctor*, so that the functor obtains a unique index in a result *View* for each overlapping pair it finds. However, the theoretical upper bound on the number intersections can exceed memory capacity when the two sets of AABBs are large, and the results *View* cannot be resized while multiple threads are in flight executing the traversal functor. Thus, we let the calling code set results capacity and allow for the possibility that the atomically updated index value could exceed it. In that case, the index value tracks the number of results that need to be stored. We dispatch the traversal functor using *parallel_reduce* and introduce a reduction variable that tracks

[7] The atomic variable is a scalar *View* whose data will be atomically read-and-updated.

[8] A well-balanced tree with N nodes has $O(\lg N)$ levels with a small constant factor.

whether the results *View* is discovered to be too small. If it is (after the *parallel_reduce* is completed), we resize the results capacity to the final value of the atomic variable, reset it, and repeat the application of the search functor via a second *parallel_reduce*.

Finally, recall that in a computational mechanics simulation, all faces and nodes can be expected to move at each timestep of the simulation. For this reason, the BVHs used for AABB-overlap searches need to be updated or recomputed each time the contact ghosting function is called. In general, we must ensure that sufficient storage is allocated for all of the sets of AABBs mentioned above. Given the need to reconstruct the BVH, it is highly desirable to use a BVH type for which there is an efficient manycore-parallel construction algorithm. We opted to use a parallel algorithm developed for efficiency on the GPU.

3.3 Morton Code Linearized BVH Trees for Intersection Search

As previously stated, the simple approach to parallelizing the top-down k-D tree construction is not efficient because there is insufficient parallelism at the top of the tree [6]. Thus, other types of BVHs and construction algorithms specifically targeting GPUs are popular research areas. We summarize the Morton code linearized tree proposed by Karras.

For many applications, including ray tracing in dynamic scenes, simulation of deforming bodies, and simulations that include large numbers of rigid bodies, the structure of a BVH computed *a priori* will not yield satisfactorily fast searches for the course of the whole runtime. The cost of the algorithm for adapting or rebuilding the BVH from scratch must be weighed against how efficiently it can be traversed. With ray tracing in mind, Lauterbach, *et al,* [27], proposed a linearized BVH construction for GPU computing that requires only one sort. Given an $2^m \times 2^m \times 2^m$ array of cells, the $3m$-bit Morton code of the 3-vector index of a cell simply interleaves the binary representations of individual indices. Thus, the Morton code for a cell is its ordinal in the Morton (or "Z order") space filling curve that traverses all the cells.

The basic idea for a Morton code linearized BVH of AABBs is to:

1. Compute the AABB that is the bounding box for all the input AABBs.
2. Compute the Morton code for the centroid of each AABB based on this bounding box.
3. Sort the Morton codes.
4. Construct an ordered radix tree with the Morton codes as keys at the leaves and the associated AABBs as data.
5. Convert the radix tree into a BVH: construct the AABB at each interior node by combining the AABBs of its children.

We implement Step 1 with a functor that resembles a simplification of the *FaceComputeBoundingBoxFunctor* mentioned above and that is applied through using *parallel_reduce*. Step 2 is performed by a centroid-computing functor applied through *parallel_for*. For the sorting step, it is useful to use a template specialization matching the computing *Device* (CPU, core, MIC, or GPU processing unit) that is chosen at compile time, as the appropriateness of different sorting algorithms [28,29,21,30–33] varies with architecture and expected problem size. In our CUDA specialization, we use the parallel *sort_by_key(..)* function from the version of the Thrust library [18] distributed in the NVidia's CUDA Toolkit.

Karras improved the speed of constructing a Morton code linearized BVH with a new algorithm for constructing a binary, ordered radix tree from a sorted set of Morton codes [6], to address Step 4. Given the Morton codes at the leaves of the tree, the algorithm computes the parent and children of each node independently and in parallel. The algorithm is designed for CUDA and trades work-efficiency for parallelism, and was implemented as a single functor dispatched through a *parallel_for*. Karras has presented performance data showing that on a Fermi GPU it enables BVH construction time similar to the overlap search time on the GPU in large scenes [35].

Following [35], we implemented Step 5 with a functor that traverses the tree upwards from its leaves. It is applied via *parallel_for*. The functor uses *atomic fetch_and_add* to check and update an array whose entries each track how many times an interior node has been visited. The first visitor of a node simply returns, because it does not know whether the AABBs of both children have been updated. The second visitor knows that they have been. The second visitor sets the AABB of the node to the result of merging the AABBs of the children, and then proceeds to visit the node's parent.

For maximum speed during search, Karras recommends computing the Morton codes for the set of AABBs for which a BVH is not computed, and sorting that set in Morton-code order. This is done to reduce divergence between threads in the same warp on the GPU during the search phase. The idea is that BVH traversals searching for intersections with respect to similarly-sized AABBs with close Morton codes can be expected to follow similar paths through the BVH.

3.4 Applying Manycore Parallel Search and BVH Construction

The first two searches in the *compute communication partners and ghosting candidates* phase are for overlaps between members of $\{\overline{B_f^q}\}$ and $\overline{B_n}$ (step (i)) and between members of $\{\overline{B_n^q}\}$ and $\overline{B_f}$ (step (ii)). These two searches do not merit the construction of a BVH for parallel search, since $\overline{B_n}$ and $\overline{B_f}$ are effectively singleton sets of AABBs. It is straightforward to parallelize the search for intersections between the members of a set of AABBs and a single AABB, and this should be done with respect to $\{\overline{B_f^q}\}$ and $\{\overline{B_n^q}\}$ only if the number of ranks is large enough.

The third and fourth searches in this phase (involving B_n and B_f) are expected to be formidable. In our tests, they each have at least 10^4 members. As a result, the search for overlaps between members of B_n and members of $\{\overline{B_f^{q\cap}}\}$ and the search for overlaps between members of B_f and members of $\{\overline{B_n^{q\cap}}\}$ are suited for parallelization. Application of Karras's manycore algorithm for BVH construction is straightforward, as is the BVH search. However, when the ratio of the size of the larger set to the smaller is large, we do not sort the larger set; we would expect the $O(N \log N)$ work in the sort to dominate the search cost for most problem scenarios

Recall that in the *communicate node boxes* phase, each rank r receives from each other rank q the set of ghosted node boxes L^q_{gnr} that intersects rank r's all-faces bounding box $\overline{B_f}$. Rank r has previously computed the set L^q_{gfs} of bounding boxes of its faces that intersect the all-nodes bounding box of rank q. The *search local faces against remote nodes* operation computes which $b^i_f \in L^q_{\text{gfs}}$ overlap any AABB in L^q_{gnr} in order to determine which faces f_i to ghost to rank q. In the original serial on-rank code for this phase, an outer loop iterates over each rank $q \in Q_{\text{rcv}}$. If L^q_{gnr} and L^q_{gfs} are both non-empty, a BVH of L^q_{gnr} is constructed, and an inner loop iterates through L^q_{gfs} and uses this BVH to search for overlaps.

We restructure the separate BVHs and searches for the different q into a single BVH construction and search in order to maximize parallelism. Conceptually, from the L^q_{gfs} and other data we construct an array of triples covering all the q and each containing:

- an AABB from L^q_{gfs};
- the ID of the corresponding face; and
- q, for which the face is a candidate to be ghosted.

We construct a similar conceptual array, $\forall q$, of pairs associating each member of L^q_{gnr} with the rank q it came from.

From these two arrays, we construct input *Views* representing possibly non-unique AABBs, copying data to the *Device*. We apply the BVH construction algorithm to the AABB `View` representation from one array, and then do an overlap search against it with the AABBs from the other in parallel. The result consists of pairs of integers that index into the two arrays. If a search result pair indexes to entries that have the same q, then the AABB of the local face overlaps the AABB of a node on rank q, and that face needs to be ghosted to rank q. Any pairs that do not satisfy this criterion are skipped.

In the current implementation, we construct the arrays for the above comparison operation and process the search results on the *Host*. After the ghost lists are constructed, the *Host* performs the ghosting of the data (via MPI communication) based on the search results, calling the Zoltan MigrateExportedData() function to prepare communication buffers and transfer the data to be ghosted.

3.5 Additional Opportunities for Manycore Parallelism in Ghosting

As mentioned in Section 1.3, `Kokkos::DualViews` are used to facilitate copying data back and forth between *Host* memory and *Device* memory. Specific categories of such data include:

- Field data, such as POSITION (displacements, etc), and
- Topology data, such as the IDs and number of nodes of a face.

Accessor functions from legacy data structures were updated to use the *Host*-side *Views*. While this staged reengineering method required significant changes across the ACME code base, the alternative of maintaining more than one copy of the entity data (one in the existing ACME data structures, and a second in device-compatible `Views`) was even less palatable.

At a higher level, the relationships among the *Device* and *Host* sides of the *DualViews* and the MPI communications network on the MPI+GPU systems we have used is shown in in Figure 6. In an MPI+X application, algorithms and data

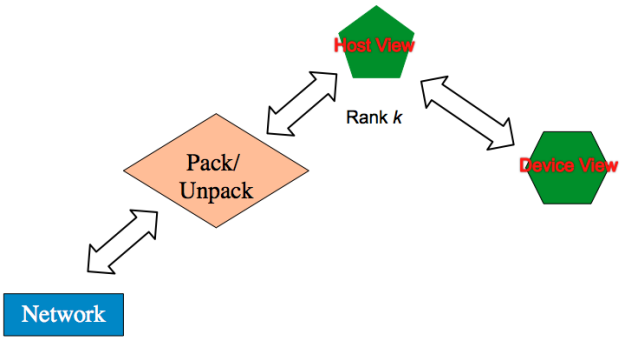

Fig. 6 Balanced data flow between host view, communications network, and coprocessor device.

structures need to be selected with respect to the efficiency

of the data flows both to the coprocessor and to the MPI communications network, the two legs of Figure 6. If either the MPI or the coprocessor data path is used out of balance with the two execution spaces, the overall performance will less than otherwise achievable.

Recall that the all-faces and all-nodes bounding boxes computed on the *Device* are transferred to the *Host* for exchange among the ranks. Here, the amount of data being sent from the *Device* to the *Host*, and then communicated using MPI is very small and likely not overly expensive. However, the `MigrateExportedData()` call at the end of the `ContactSearch::DoGhosting_New_NodeFace` function is quite expensive. In the migration operation, for each MPI rank with which the current rank needs to send ghost entity information, there may be thousands of entities to be migrated. The bulk of time spent in `MigrateExportedData()` is assembling and connecting various ACME data structures on the *Host*. Potentially, parallel algorithms could perform this work faster on the *Device* if more of the underlying ACME data structures are restructured. If the computation needed to form the communication buffers was performed on the *Device*, and the communications architecture supported direct buffer sends from the device address space, a design such as the one illustrated in Figure 7 would result. This model would avoid the need for data transfer from the *Device* to the *Host* and the host migration processing and communications operations.

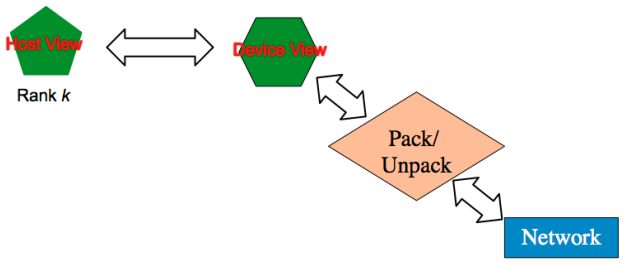

Fig. 7 Data flow between host view and coprocessor device where all processing occurs on the device.

In Contact Global Search, the all-nodes and all-faces bounding boxes, the ghosting node boxes, and the faces to be ghosted that are calculated on the *Device* could be packed in place on the *Device* . The packed data could then sent over the network using an MPI all-to-all via the *Device* buffer. Note that a bulk copy between *Host* and *Device* memory is often preferable to serial packing/unpacking on the *Host*. These AABBs and ghosted faces (including their field data) would similarly be received directly on the *Device* for use in the following search operations. More detailed investigation of this approach is left for future work.

4 Performance Measurements

We compared the performance of the new contact ghosting function against the ACME reference implementation on "brick wall" test problems employing 278K, 1113K, 4458K, 7820K, and 17818K faces. We ran the code on 4 MPI ranks on Shannon, with one MPI rank per compute node. We used one regular processor core (*Host*) and one GPU (*Device*) per MPI rank.

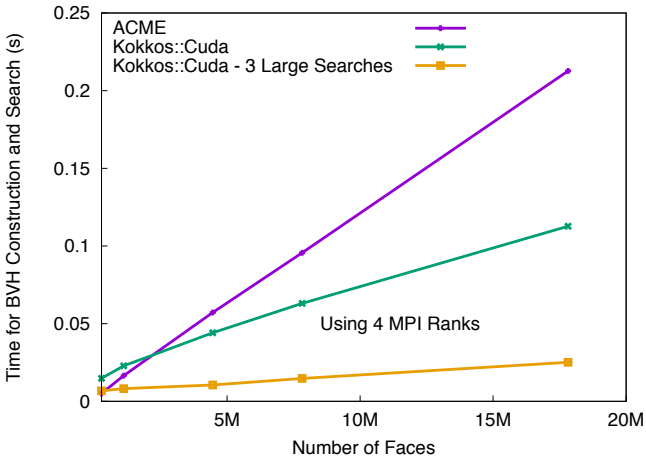

Fig. 8 Scalability of the Morton code linearized BVH construction and intersection search algorithm compared with the reference ACME implementation. Both curves use the same MPI decomposition (4 ranks). The GPU-executed linearized BVH algorithm is significantly faster and scales better than the reference implementation.

Figure 8 shows the scaling of the Morton code linearized BVH construction and intersection search algorithm compared with the ACME implementation. We include all five intersection searches described in Section 3.4 plus the time required to read and write data to arrays for communication and registering faces for ghosting[9]. The upper curve is the reference ACME implementation using 4 MPI ranks. The middle curve shows the new implementation where the GPU on each rank executes the parallel Morton code linearized BVH construction and search algorithm. The MPI decomposition is identical in both cases. Recall that not all of this code is parallel, only the three searches that process a large number of bounding boxes are run in parallel on the GPU. The bottom curve shows the scalability of these three large searches, ignoring the two serial searches.

The gap between the top curve and the middle curve represents the overall performance gain in the global search due to the proposed parallel BVH construction and search approach. Note that the middle curve includes the filtering step on the *Host* that follows the search in the modified *search*

[9] These operations are included in the outer loops of the respective searches in the ACME reference implementation.

local faces against remote nodes operation, since the original ACME control flow did not require filtering. (*c.f.* Section 3.4.) Note that the parallel search does extra work compared to the reference, serial version. First, for each face box $b_f^i \in L_{\text{gfs}}^q$, the parallel algorithm actually finds *all* (off-rank) AABBs in L_{gnr}^q that overlap b_f^i, whereas the serial algorithm can move on to the next face box once it has found *any* such node box from rank q. In our test problems, this resulted in extra BVH traversal work that identified as many as three times as many overlapping pairs as the serial, reference code. Second, when a face box overlaps more than one node rank bounding box, the face box will be included on multiple occasions in the search (recall that this is the reason for the filtering step). Furthermore, we made no attempt to estimate the capacity for the search results buffer and allowed it grow incrementally as needed throughout the contact ghosting function. As a result, all three searches repeated the BVH traversal phase when running the test problems, as the first pass was used only to determine the necessary capacity of the array required (*c.f.* Section 3.2.)

Comparison of the gap between the top and middle curves against the bottom curve shows excellent performance of the GPU implementation of the linearized BVH algorithm. We would expect the algorithm to continue to scale well as problem size on the GPU increases, up to the point of running short of memory to hold the on rank search entities within the GPU memory space. Further work will be needed before the search algorithm is robust for analysis applications, as it will be necessary to modify either the number of MPI ranks or the entity decomposition to prevent overfilling the GPU memory space with search entities as the simulation proceeds.

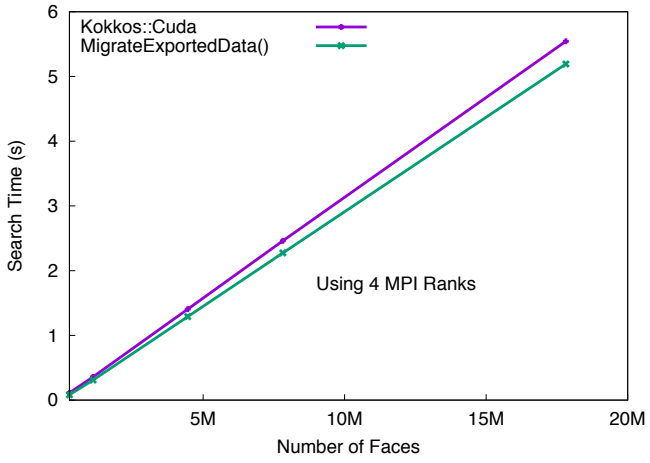

Fig. 9 Scalability of the contact ghosting function, illustrating the cost of migrating the entities calculated withing the global search, MigrateExportedData(). The search time is becoming negligible with respect to the time required to migrate the entities in the ghosting operation. Further acceleration of the search algorithm will require a strategy to decrease the MPI intercommunication time, as suggested in Figure 7.

Figure 9 compares the time required by the contact global (ghosting) search function to that required to migrate the ghosts using MPI. Again, all cases employ 4 MPI ranks, and each MPI rank has its own GPU. It is clear from this result that further improvement of the search algorithms will result in little additional performance, as the MPI communication time now monopolizes the overall contact search and ghosting process. Further improvement will depend on increasing the efficiency of the data transfer from GPU to GPU across the MPI communications channel.

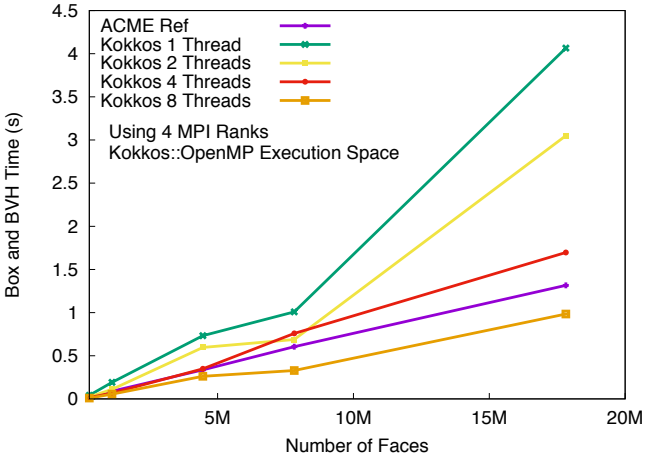

Fig. 10 Scalability of the contact ghosting function on a typical multicore shared memory multiprocessor (SMP). We compare the parallel box construction and Morton code linearized BVH algorithm using multiple threads (1, 2, 4, and 8) with 4 MPI ranks, varying the number of active CPU cores per rank. This result uses the `OpenMP` Kokkos device, and shows that Kokkos provides a portable implementation on both GPU and multicore devices.

To examine the portability of the new ghosting search algorithm, we next considered conventional multi-core SMP, using the Kokkos `OpenMP` execution space. For the purposes of portability, we used a simple parallel radix sort in the BVH construction algorithms. Figure 10 shows how the ghosting function performance scales on the same machine used for the GPU runs, but employing up to 8 Sandy Bridge cores per MPI rank. Again, we employ 4 MPI ranks with an identical domain decomposition as before. The data indicates that as the number of cores were increased, there was a significant improvement in overall performance. However, the single-thread ACME reference implementation was more efficient than all but the 8 core result. Recall that this is due to the fact that the parallel implementation performs significantly more work than the reference implementation. Indeed, one would not expect that the single-thread OpenMP performance would compare with that of the (non-threaded) reference ACME implementation.

Note that since the Morton code algorithm presented here is tailored for the GPU, it is likely not well structured

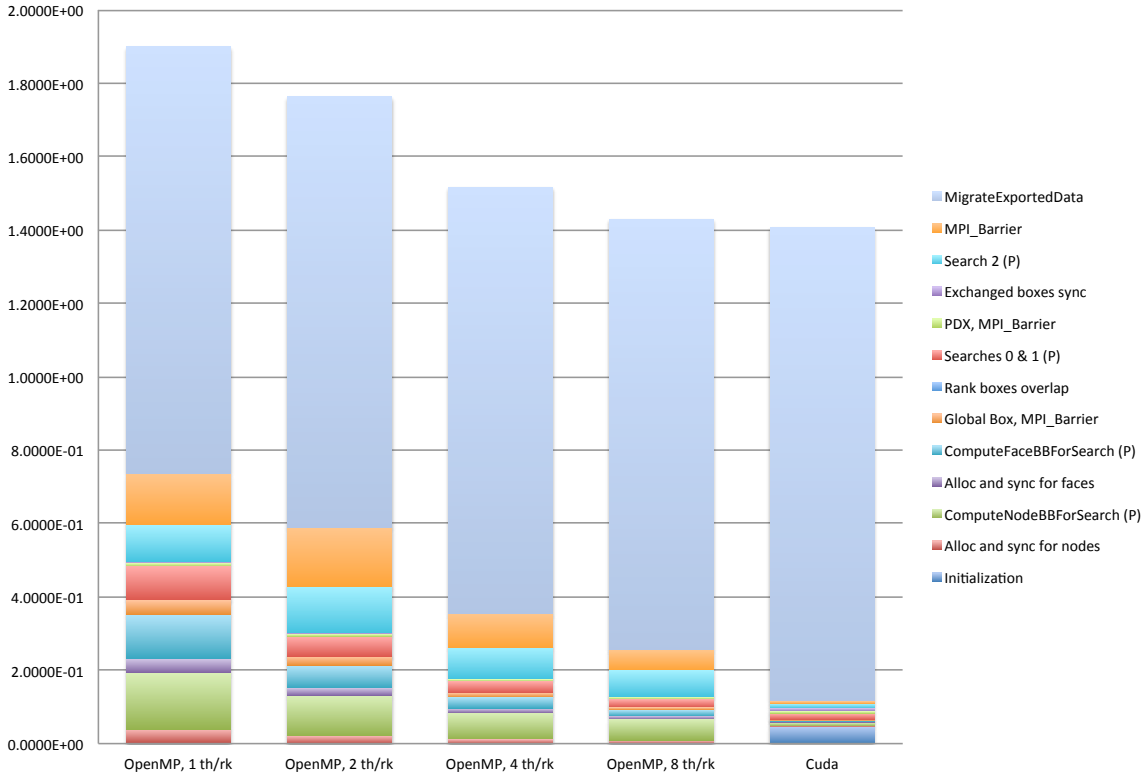

Fig. 11 Comparison of the overall effort required to execute the contact ghosting operation on the 4458K face problem. These results are for Kokkos parallel implementation using the CUDA execution space on 4 GPUs (one per rank) and the OpenMP execution space (with 1, 2, 4, and 8 threads) on an SMP, also using 4 ranks.

for multicore SMP architectures. Figure 11 shows the timings for the 4458K face problem using a Kokkos::OpenMP execution space on an SMP with 1, 2, 4, and 8 OpenMP threads per MPI rank, adjacent to the Kokkos::Cuda compilation on a GPU co-processor. Enhancing the performance and scalability of the search algorithm on an SMP is left to future work, it is quite likely that an improvement in performance could be obtained. However, such an improvement may come at the cost of GPU performance. It remains to be demonstrated if is is possible to obtain near optimal performance of contact global search across diverse architectures, perhaps spanning SMPs, GPUs, and the INTEL MIC, using a single-source implementation with Kokkos serving as the abstraction layer.

In Figure 11, 13 individual timers were used to study the various contact ghosting operations. The timers' spans partition the contact global search function. Four of these sections (marked with a P) correspond to phases of the contact ghosting function algorithm that we attempted to accelerate using parallelization:

- Computing the expanded bounding boxes for the local nodes on each MPI rank.
- Computing the expanded bounding boxes for the local faces on each MPI rank.
- Searching for intersections between local node boxes and rank wide faces boxes, and between local face boxes and rank wide node boxes. One timer ("Searches 0 & 1") covers both searches, which are contiguous in the code.
- The box intersection search ("Search 2") that is applied to the results of those two searches, following the parallel data exchange.

Other phases cover initialization, data allocation and initialization, synchronizing data between the *Host* and *Device* (for CUDA/GPU), parallel data exchanges, MPI Barriers, data migration, and sections too insignificant to merit parallelization in the 4 MPI rank example. Note that the timings of phases that involve MPI Barriers and synchronization are sensitive to skew in the respective previous phases.

The single thread OpenMP results show that all four of the phases in which we introduced parallelization consume a significant enough amount of time and could potentially benefit from parallel execution. The face box computation shows good strong scaling on the OpenMP *Device*. Indeed, when the number of OpenMP threads were increased from 1 to 8, this operation was accelerated by a factor of 7. The first two searches show significant improvement in performance, and are more than 4 times as fast with 8 threads compared to a single thread. The node box computation scales less well. Using 8 threads it about 2.6 as fast than on a sin-

gle thread. The difference between the node box functor and the face box functor is that the former uses an atomic variable that each thread increments to obtain space in a results array. While the GPU provides high performance atomics using CUDA, the performance of the atomic operation on OpenMP results in a bottleneck. The searches also make use of atomics, although to a lesser proportion to the amount of computation done.

The final search phase is only slightly faster using 8 threads as on a single thread. Overall, the performance of the algorithm on OpenMP is respectable given the additional overhead involved in the parallel search; performance is slightly slower on 2 threads than 1 and slightly faster on 4 threads than 1.

Finally, we observe that initialization of the search is has appreciable cost on the GPU but is insignificant in the case of OpenMP. We focused the code optimization on the GPU platform, but clearly more work can be done in this area to further improve GPU performance. Work is also currently underway to study this algorithm on the Intel Xeon Phi coprocessor.

5 Conclusions

The paper details the development of a contact global (ghosting) search formulation to make use of manycore parallelism. Our primary target was MPI+GPU architectures. While some aspects of this work that lead to substantial performance gains are straightforward, for intersection searches involving sets of bounding boxes we leveraged capabilites developed by researchers at NVIDIA targeted at visualization operations on a GPU. Here, we employ a Morton code linearized BVH search algorithm and a BVH construction algorithm advanced by Karras [6,7]. This algorithm, if it were applied in the most straightforward manner, could require performing a number of searches linear in the number of MPI ranks. As this would limit the benefits of manycore parallelism, we modified the global search algorithm to combine all but a constant number of searches into a single search operation.

We implemented the contact search algorithm using Sandia National Laboratories' Kokkos multicore hardware abstraction package, to allow the use of the search capability in a portable manner across GPU and multicore hardware. We chose to demonstrate this algorithm inside of Sandia's ACME (Algorithms for Contact in a Multiphysics Environment), taking advantage of the set of correctness tests available in ACME to ensure correct operation of the final code. Our study of developing MPI+GPU/multicore search capabilities and implementing them in ACME followed these steps:

- To select what sections of code to address, we first used profiling to identify where to focus ACME code and data access restructuring activity. We then looked for the top time consuming sections while considering dependencies and the existing code structure.
- We replaced the original ACME contact ghosting function with an implementation that builds on Kokkos for portable on-rank parallelization. Changes in the new version include:
  a) parallel compution of node and face bounding boxes,
  b) implementation of the parallel Morton code linearized BVH algorithm using Kokkos, and
  c) calling the parallel methods using generic `Kokkos::DualView` interfaces and functors.
- We compared the final results on both GPU and multicore hardware to assess the degree to which Kokkos has enabled code portability and convenience. Initial results of the approach demonstrate very good performance on GPUs, and significant parallel performance on multicore architectures.

Future work includes improving the scalability of the algorithm on `OpenMP` and `PThreads` and developing results on the Xeon Phi. Use of the "count, allocate, and fill" pattern [16] for returning parallel algorithm results would remove some performance limitations caused by the use of atomic operations. In addition, redesigning the final entity migration communication could take advantage of coprocessor execution and direct buffer communications between coprocessors. Overall performance of the contact implementation could be further increased by performing the contact local search operation given the resident entity data migrated between the coprocessors during that operation. Finally, it should be possible to also perform the contact enforcement phase subsequent to the local search, without appreciable data transfer between the *Host* and *Device*.

The Kokkos hardware abstraction library has appeared to be suitable for this study. One goal of Kokkos is to provide *performance portability*, where an algorithm implementation using the Kokkos library is portable across a variety of architectures and the implementation provides performance near that achievable with a natively-coded version of the method on the hardware of interest. This study demonstrated source code portability across both multicore and GPU architectures. It has found good performance with little work where the chosen algorithm was appropiate for the hardware. We did not compare the Morton code linearized BVH search implementation with architecture specific versions, and additional study would be required to verify the performance dimension of the Kokkos implementation. Practically, at least in the case of the GPU, additional performance in the search would not increase the performance of the overall contact implementation as the time spent in entity migration overwhelms the rest of the search. It was straight-

forward to implement the Morton code linearized BVH construction and search algorithms using Kokkos, and to use the Kokkos portable data structures to redesign the ACME code to efficiently transfer data between the *Host* and *Device*.

This study begins the path of developing a performance portable contact library, building on the excellent results obtained with the Morton code linearized BVH algorithm and straightforward parallelized bounding-box computation. The performance gains from these algorithms overcome additional costs within contact ghosting accrued on the *Host* due to CUDA optimized data layouts in Kokkos data structures. Kokkos versions of local search and contact enforcement would be a natural evolution of the demonstrated capability that could take advantage of coprocessor data structures developed during this effort.

Acknowledgements

This work was funded by the U.S. Department of Energy through the NNSA Advanced Scientific Computing (ASC) Integrated Codes (IC) program.

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

The authors would like to thank Tero Karras of NVIDIA Corporation for his CUDA code and suggestions.

References

1. Glen Hansen. A Jacobian-free Newton Krylov method for mortar-discretized thermomechanical contact problems. *J. Comput. Phys.*, 230:6546–6562, 2011.
2. Kevin H. Brown, Micheal W. Glass, Arne S. Gullerud, Martin W. Heinstein, Reese E. Jones, and Thomas E. Voth. ACME: Algorithms for contact in a multiphysics environment API version 2.2. Technical Report SAND2004-5486, Sandia National Laboratories, 2004.
3. A. Khamayseh and G. Hansen. Use of the spatial *k*D-tree in computational physics applications. *Comm. Comput. Phys.*, 2(3):545–576, 2007.
4. S. W. Attaway, B. A. Hendrickson, S. J. Plimpton, D. R. Gardner, C. T. Vaughan, K. H. Brown, and M. W. Heinstein. A parallel contact detection algorithm for transient solid dynamics simulations using PRONTO3D. *Computational Mechanics*, 22:143–159, 1998.
5. Karen Devine, Erik Boman, Robert Heaphy, Bruce Hendrickson, and Courtenay Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering*, 4(2):90–97, 2002.
6. Tero Karras. Maximizing parallelism in the construction of BVHs, octtrees, and *k*-d trees. In C. Dachsbacher, J. Munkberg, and J. Pantaleoni, editors, *Eurographics/ACM SIGGRAPH Symposium on High Performance Graphics*, pages 33–37, 2012.
7. Tero Karras. Thinking parallel, part II: Tree traversal on the GPU. `http://devblogs.nvidia.com/parallelforall/thinking-parallel-part-ii-tree-traversal-gpu/`, 2012.
8. OpenMP application program interface, version 4.0, July 2013.
9. David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.
10. NVIDIA Corporation. CUDA C programming guide. `http://docs.nvidia.com/cuda`, March 2015.
11. Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. OpenACC: First experiences with real-world applications. In *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, pages 859–870, Berlin, Heidelberg, 2012. Springer-Verlag.
12. Arch D. Robison. Composable parallel patterns with Intel Cilk Plus. *Computing in Science and Engineering*, 15(2):66–71, 2013.
13. James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.
14. Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. In *24th ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA'09), Orlando, FL*, 2009. Also appeared in Sigplan Not., **44**(10) 227–242, 2009.
15. H. Carter Edwards and Christian R. Trott. Kokkos: Enabling performance portability across manycore architectures. Boulder, CO, 2013. XSEDE. `https://www.xsede.org/documents/271087/586927/Edwards-2013-XSCALE13-Kokkos.pdf`.
16. H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos, a manycore device performance portability library for C++ HPC applications. San Jose, CA, March 2014. GPU Technology Conference. `http://on-demand.gputechconf.com/gtc/2014/presentations/S4213-kokkos-manycore-device-perf-portability-library-hpc-apps.pdf`; also Sandia National Laboratories SAND2014-2317C.
17. H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, December 2014.
18. Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for CUDA. *GPU Computing Gems Jade Edition*, page 359, 2011.
19. A. Robinson et al. ALEGRA: an arbitrary Lagrangian-Eulerian multimaterial, multiphysics code. In *Proc. 46th AIAA aerospaces sciences meeting*, 2008.
20. S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof a call graph execution profiler. In *Proc. ACM SIGPLAN '82 Symposium on Compiler Construction*, pages 120–126, June 1982.
21. Michael McCool, Arch Robison, and James Reinders. *Structured Parallel Programming*. Morgan Kaufmann, 2012.
22. G. E. Blelloch. Scans as primitive parallel operations. *IEEE Trans. Computers*, 38(11):1526–1538, November 1989.
23. Message Passing Interface Forum. MPI: a message-passing interface standard. Technical report, Knoxville, TN, USA, 1994. Available via http://www.mpi-forum.org.
24. OpenMP application program interface, version 1.0, October 1997.
25. C. R. Trott, M. Hoemmen, S. D. Hammond, and H. C. Edwards. Kokkos: The programming guide. Technical Report SAND2015-4178 O, Sandia National Laboratories, 2015. Available at `https://github.com/kokkos`.
26. Intel Corporation. Intel threading building blocks reference manual, 2015. `https://www.threadingbuildingblocks.org/docs/help/reference/`.
27. C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast BVH construction on GPUs. *Computer Graphics Forum*, 28:375–384, 2009.
28. Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. Fast sort on CPUs and GPUs: A case for bandwidth oblivious SIMD sort. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 351–362, New York, NY, USA, 2010. ACM.

29. Andrew Davidson, David Tarjan, Michael Garland, and John D. Owens. Efficient parallel merge sort for fixed and variable length keys. In *Innovative Parallel Computing*, page 9, May 2012.
30. Arch D. Robison. A parallel stable sort using C++11 for TBB, Cilk Plus, and OpenMP. `https://software.intel.com/en-us/articles/a-parallel-stable-sort-using-c11-for-tbb-cilk-plus-and-openmp`, April 2014.
31. Linh Ha, Jens Krüger, and Cláudio T. Silva. Fast four-way parallel radix sorting on GPUs. *Computer Graphics Forum*, 28(8):2368–2378, 2009.
32. Hagen Peters, Ole Schulz-Hildebrandt, and Norbert Luttenberger. Fast in-place sorting with CUDA based on bitonic sort. In *Proceedings of the 8th International Conference on Parallel Processing and Applied Mathematics: Part I*, PPAM'09, pages 403–410, Berlin, Heidelberg, 2010. Springer-Verlag.
33. Erik Sintorn and Ulf Assarsson. Fast parallel gpu-sorting using a hybrid algorithm. *J. Parallel Distrib. Comput.*, 68(10):1381–1388, October 2008.
34. N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore gpus. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10, May 2009.
35. Tero Karras. Thinking parallel, part III: Tree construction on the GPU. `http://devblogs.nvidia.com/parallelforall/thinking-parallel-part-ii-tree-traversal-gpu/`, 2012.