Minimizing Development Costs for Efficient Many-Core Visualization Using MCD^3

Kenneth Moreland[a], Robert Maynard[b], David Pugmire[a], Abhishek Yenpure[c], Allison Vacanti[b], Matthew Larsen[d], Hank Childs[c]

[a]*Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA*
[b]*Kitware, Inc., Clifton Park, New York, USA*
[c]*University of Oregon, Eugene, Oregon, USA*
[d]*Lawrence Livermore National Laboratory, Livermore, California, USA*

Abstract

Scientific visualization software increasingly needs to support many-core architectures. However, development time is a significant challenge due to the breadth and diversity of both visualization algorithms and architectures. With this work, we introduce a development environment for visualization algorithms on many-core devices that extends the traditional data-parallel primitive (DPP) approach with several existing constructs and an important new construct: meta-DPPs. We refer to our approach as MCD^3 — **M**eta-DPPs, **C**onvenience routines, **D**ata management, **D**PPs, and **D**evices. The twin goals of MCD^3 are to reduce developer time and to deliver efficient performance on many-core architectures, and our evaluation considers both of these goals. For development time, we study 57 algorithms implemented in the VTK-m software library and determine that MCD^3 leads to significant savings. For efficient performance, we survey ten studies looking at individual algorithms and determine that the MCD^3 hardware-agnostic approach leads to performance comparable to hardware-specific approaches: sometimes better, sometimes worse, and better in the aggregate. In total, we find that MCD^3 is an effective approach for scientific visualization libraries to support many-core architectures.

Keywords: scientific visualization, many-core architectures, data-parallel primitives

1. Introduction

As supercomputers increasingly include many-core architectures, visualization software design must adapt to support these architectures. This is a significant challenge, since existing parallel visualization software, like ParaView [1] and VisIt [2], have primarily focused on MPI-only parallelism. Further, these efforts represent hundreds of person years of effort and contain hundreds of algorithms. As a result, new, many-core capable designs must achieve twin goals: efficient performance and small development time. This latter goal is particularly necessary not only because of the large number of visualization algorithms but also because of the large number of potential hardware architectures. The worst case would require an implementation for every possible pair of algorithm and architecture. Instead, a better scenario is to achieve "portable performance," i.e., a single, hardware-agnostic implementation for each algorithm that works efficiently on all architectures.

Data-parallel primitives [3] (DPPs) have the potential to achieve the twin goals of efficient performance and small development time, but their application in the visualization space is non-trivial. In particular, the nature of scientific data (mesh-based data) incurs extra burden with DPPs, as most DPPs are designed to operate on arrays of data and are not concerned with issues such as which vertices lie within a cell. This has two unfortunate consequences for implementing visualization algorithms with DPPs. First, these complex data structures add an increased indexing burden on the developer, who has to maintain and follow links in the data. Second, the irregular data typical of visualization algorithms means that the products of a visualization algorithm do not always have a one-to-one relationship with the input. Some inputs contribute nothing to the output whereas other inputs generate multiple things, creating a "jagged" access pattern for the inputs and outputs of a visualization algorithm. As a result, using DPPs could cause more harm than good — visualization algorithm developers could spend more time addressing data model issues than they would gain from DPPs' benefits.

With this paper, we introduce a design that augments DPPs with several constructs to shield visualization algorithm developers from potential pitfalls. Our design incorporates existing data management practices that separate memory layout from execution space, adding support for many data layouts and mesh types. We also incorporate the common practice of providing convenience routines for common operations, with our unique contribution being a selection of routines useful for scientific visualization algorithms, such as locating which cell contains a point or such as finding the minimum and maximum value for a field. Finally, and most importantly, we introduce a new construct, which we call meta-DPPs, which combine DPPs and data management to address issues such as following links in data and jagged access patterns. In all, we refer to our algorithm development environment as **MCD^3** — **M**eta-DPPs, **C**onvenience routines, **D**ata management, **D**PPs, and **D**evices.

The fundamental research questions of this paper are on the

Email address: hank@uoregon.edu (Hank Childs)

overall viability of the MCD^3 approach for many-core visualization and on evaluating the twin goals of minimizing development costs while providing efficient execution times. These questions are explored by analyzing the VTK-m library [4], which is an open source effort that is using MCD^3 for many-core visualization. In all, the contributions of this paper are:

- New meta-DPPs designed for common patterns for scientific visualization algorithms;
- Evaluation of developer costs by studying 57 visualization algorithms and their usage of MCD^3;
- Evaluation of efficient execution time by surveying 10 performance studies using MCD^3 or DPP-based visualization; and
- Evaluation of overall efficacy of MCD^3 by studying the use of MCD^3 elements in visualization code.

Finally, these contributions combine to form our overall finding: the MCD^3 approach is effective for creating a portably-performant many-core visualization library.

2. Motivation, Background, and Related Work

This section is divided into three parts. First, Section 2.1 summarizes background on scientific visualization on supercomputers, and provides motivation for MCD^3's goals of minimizing developer time and efficient support for many-core architectures. Next, Section 2.2 provides background on data-parallel primitives. Finally, Section 2.3 surveys the works most closely related to our own research.

2.1. Scientific Visualization on Supercomputers

Scientific visualization software is an important component in many computational science workflows on supercomputers. In these workflows, simulations produce large data sets (up to trillions of cells per time step), meaning visualization software must be able to process such data as well. Although there are multiple strategies for processing large data sets, such as subsetting and multi-resolution approaches, the most common approach is parallel processing — the supercomputer that generated a data set can then be used to visualize that data set. That said, execution time remains a significant concern, since some visualization algorithms require significant computation and/or are difficult to parallelize, especially in the context of very large data sets [5].

Scientific visualization software has great potential for scaling to large numbers of users, because of the commonality in the data produced by computational simulations (mesh-based or point data) and the algorithms they employ (isosurfacing, slicing, volume rendering, streamlines, etc.). In all, building up a software instance to support many algorithms, features, operating systems, hardware platforms, file formats, etc., allows a handful of visualization scientists to make software that can be useful for many domain scientists (i.e., thousands to tens of thousands). This pattern of visualization software scaling to large groups of users has occurred many times, including libraries like AVS [6], OpenDX [7], and VTK [8], and end-user tools like EnSight [9], FieldView [10], MegaMol [11], ParaView [1], SCIRun [12], VAPOR [13], VisIt [2], and yt [14], just to name a few. This history provides motivation that developing a richly-featured many-core visualization library has the potential to provide much greater impact than developing multiple bespoke solutions for different stakeholder groups.

Modular designs are a key strategy for many visualization software packages, since they reduce development costs when implementing a rich feature set. Typically, each module encodes a single visualization algorithm. These modules can then be composed together to meet the needs of domain scientists. This modular design is enabled by data models, i.e., an in-memory description of the representation of meshes and fields. A data model allows the modules to create outputs that can be used as inputs to other modules. Modular designs have enabled some visualization software packages to grow to hundreds of modules. That said, the nature of these algorithms is quite diverse, which places a burden on underlying frameworks to support diverse processing demands.

Finally, visualization algorithms are regularly being run using in situ processing [15, 16], i.e., visualizing data as it is generated. In situ processing has become increasingly necessary because supercomputer disks are unable to store and load simulation data quickly enough [17, 18]. In situ avoids I/O, thus sidestepping this bottleneck. That said, the technique requires that visualization algorithms are run on the supercomputer, typically by sharing access to simulation data and hardware resources. In all, this motivates the need for visualization to run efficiently on many-core architectures, as supercomputers increasingly contain such architectures. Further, a recent Dagstuhl workshop on in situ processing identified efficiency on exascale machines as one of in situ's top challenges [19].

2.2. Data-Parallel Primitives

Guy Blelloch's Ph.D. dissertation [3] considered parallel vector models as a way to unify theory, language, and architecture. In this work, he introduced two important primitives, scan and segmented instructions, and demonstrated that diverse tasks could be parallelized using these primitives. These two primitives are part of a larger group of primitives referred to as "data-parallel primitives" (DPPs).

DPPs operate on arrays (vectors) of data. They need to operate quickly when supplied with sufficient parallelism — if an array is of size N and if there are at least N cores, then a DPP needs to complete in $\mathcal{O}(\log N)$ time. Many operations can achieve this time bound, including Reduce, Scan, Transform, and Sort.

DPP-based programming involves selecting the series of primitives needed to carry out the desired task. Further, some DPPs can be parameterized with custom operations. For example, the Reduce DPP accepts a binary operator indicating how data is combined, which allows it to be modified to compute the sum, find the minimum or maximum, or other operations. This custom operation typically comes as a lightweight function that operates on one or two elements in the data array. These functions often come in the form of "functors," which is an object that can be called like an ordinary function. In this paper, we

use the term functor to refer to some type of function passed to a generic routine like a DPP.

DPPs provide multiple benefits for programmers. One benefit is in performant algorithms. Programmers can only choose from operations that are $\mathcal{O}(\log N)$ or better (given sufficient parallelism), which eliminates scenarios where implementations have subtle performance issues that blow up at scale. Further, this model improves performance by sufficiently decomposing work such that each core on a many-core device can participate. A second benefit is increased reliability, as this model eliminates common thread safety issues. The third and biggest benefit from a DPP-based approach is its hardware-agnostic nature: portability, portable performance, and future proofing are all inherent. DPP programming makes no assumptions about the underlying hardware. The majority of a porting task is implementing the individual DPPs (e.g., Reduce, Scan, etc.) efficiently for a given architecture whereas little to no porting is necessary for algorithms based on the DPPs. Further, a programmer using DPPs does not need to learn details for CUDA, HIP, etc., since DPPs provide a layer over these environments. Because of these benefits, the DPP approach has been steadily gaining in popularity, including packages such as AMP [20], BOLT [21], Boost.Compute [22], and Thrust [23], as well as concepts in a recent C++ standard.

2.3. Scientific Visualization on Many-Core Architectures

Scientific visualization software has a rich history of using GPUs, not only for rendering, but also for carrying out the visualization algorithms and transformations to create renderable forms. For more information on this topic, we refer the reader to the following excellent works (in chronological order): Weiskopf's book [24], and surveys by Ament et al. [25], Rodriguez et al. [26], and Beyer et al. [27].

In the early 2010's, three scientific visualization libraries emerged that focused on portable performance for many-core architectures. While all three had similar elements, each had a special focus: Dax [28] on execution models, EAVL [29] on data models, and PISTON [30] on algorithms. Ultimately, these three projects merged to become VTK-m [4], and the strengths of each were brought forward to the new project. VTK-m is named to also evoke the positive elements of the Visualization ToolKit (VTK) [8], i.e., rich feature set and open source with a large developer community, with the 'm' denoting support for many-core architectures. VTK-m is foundational to this paper, as it provides an implementation of the MCD^3 development environment and thus can be used for our evaluation. In particular, our evaluation depends on studying the implementations of its 57 algorithms as well as 10 performance studies on individual algorithms from VTK-m or its predecessors.

Finally, there are several notable efforts on many-core scientific visualization environments. On the whole, these efforts are aimed at delivering capability directly to domain scientists, where the focus on MCD^3 is on delivering an environment to visualization algorithm developers. Further, previous work has had a reduced emphasis on central issues for MCD^3 — jagged access patterns, following links in unstructured data, etc. We discuss four prominent packages, each of which used domain-specific languages (DSLs) as part of their approach for supporting many-core architectures. First, Scout [31, 32] was an early project in this space, providing GPU support and delivering operations such as volume rendering and derived field generation, among others. Second, Diderot [33, 34] placed special emphasis on tensor data and mathematical operations, as well as biomedical images. Third, ViSlang [35] provided an OpenCL-based system that delivered a variety of DSLs, each of which are referred to as "Slangs." Fourth, Vivaldi [36] placed a special focus on distributed heterogeneous systems and demonstrated volume rendering and image segmentation operations.

3. MCD^3 Design

The MCD^3 design relies on five constructs:

- **Devices**, which enable code to run on a given hardware architecture.
- **DPPs**, which provide parallel processing patterns.
- **Data management**, which insulates algorithms from data layout complexities. These complexities range from how data is organized (e.g., structure-of-arrays vs array-of-structures) to different types of meshes (e.g., unstructured, rectilinear, etc.) to different memory spaces (e.g., host memory, device memory, or unified managed memory).
- **Meta-DPPs**, which are parallel processing patterns that involve one or more DPPs. The word choice of "meta" is meant to evoke its definition of "denoting something of a higher or second-order kind."
- **Convenience routines**, which encapsulate common operations for scientific visualization.

Differentiating between the last two constructs, meta-DPPs are effectively templates for parallel processing patterns, while convenience routines are essentially sub-routines.

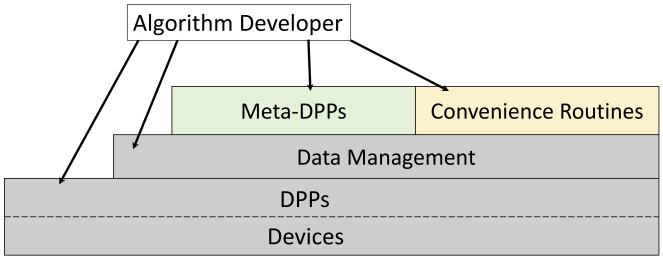

Figure 1: Diagram showing the layered natures of MCD^3. Gray boxes (data management, DPPs, and devices) represent existing constructs that we have incorporated into our MCD^3 design, while colored boxes represent new contributions. In particular, meta-DPPs are a new construct, while convenience routines represent a new contribution through the specific visualization routines we have identified as useful across a wide array of algorithms.

Figure 1 shows how the constructs in MCD^3 interact. Meta-DPPs and convenience routines work through a data management layer to call actual DPPs. DPPs then provide hardware-specific implementations through the device layer. Each layer

leads to savings in development time, and these savings are multiplicative across layers — if a meta-DPP represents the work of N_1 DPPs, if the data management layer provides support for N_2 data layouts, and if the DPP/device layer provides support for N_3 devices, then one MCD^3 algorithm can do the work of $N_1 \times N_2 \times N_3$ non-MCD^3 implementations. That said, this analysis should be considered as simply an upper bound, as developers in non-MCD^3 settings would not implement the cross product of options, but rather only those that they need. Regardless, the MCD^3 approach makes this cross product available from inception.

The remainder of this section is organized into four parts. It begins by describing algorithm development using MCD^3 (Section 3.1). It next describes our five proposed meta-DPPs (Section 3.2) and our five proposed convenience routines (Section 3.3). It then concludes with a summary of the constructs we use from previous work (data management, DPPs, and devices) and how we use them in MCD^3 (Section 3.4). Finally, an example algorithm in MCD^3 can be found in our supplemental material.

3.1. MCD^3 Algorithm Development

MCD^3 algorithms operate by applying a sequence of operations. Formally, an MCD^3 algorithm is composed of N operations ($N \geq 1$), which can be labeled S_1 through S_N. Each operation S_i comes from one of MCD^3's three main programming constructs: meta-DPPs, DPPs, and convenience routines. For example, operation S_i may be "Reduce," which is a DPP. Further, operations may be used repeatedly, e.g., S_i, S_j, and S_k all may be "Reduce." The inputs and outputs of each S_i operation vary, and can be single numbers, arrays, involve mesh topology, etc. An algorithm executes by invoking operation S_1, and then S_2, and so on. The inputs to operation S_i may include outputs from any of the previous operations (S_1 up to S_{i-1}) as well as inputs to the algorithm. Similarly, the outputs from operation S_i may be used as inputs for any of the subsequent operations (S_{i+1} to S_n) as well as outputs for the algorithm. The operations execute sequentially, meaning that S_{i+1} is invoked only after S_i completes.

MCD^3 algorithms operate on a combination of host and device. An algorithm developer will only write host code. However, when an algorithm developer invokes an operation (i.e., some S_i) on the host, this cues the MCD^3 infrastructure to execute code on the device using parallel code. Finally, if an operation generates an array, then the array is stored in the device's memory, i.e., not transferred back to the host. This array can then be accessed by subsequent operations. In all, data is rarely transferred from device to host.

MCD^3-based algorithm development strongly resembles DPP-based algorithm development. The job of an algorithm developer is to devise a sequence of operations to carry out their desired task(s). Challenges with this job include determining which patterns best enable the task to be performed efficiently, as well as ensuring that the outputs of some operations are in the right format to be inputs for subsequent operations. The key distinctions for our MCD^3 design, however, are: (1) the addition of a new programming construct (meta-DPPs), (2) the presence of convenience routines for scientific visualization, and (3) a data management layer that deal with issues ranging from data layout to different types of meshes to the jagged nature of inputs and outputs.

3.2. Meta-DPPs

Meta-DPPs are the most important part of the MCD^3 system. They identify what type of data to parallelize over, e.g., parallelizing over cells, over points, or over elements in an array. Meta-DPPs then provide infrastructure that organizes input data (arrays or data sets) for execution on many-core devices based on the parallelization type (cells, points, elements in an array). On the algorithm developer side, the developer provides a functor to carry out the operation for a single instance. For example, if the meta-DPP parallelizes over cells, then an algorithm developer would write a functor that operates on a single cell. Further, the meta-DPP would (for the example of parallelizing over cells) identify which points are incident to a cell and provide (for example) their spatial locations and field values to the functor.

In all, meta-DPPs lower the burden for implementing visualization algorithms, since algorithm developers are freed from worrying about parallelization and data reorganization. Of course, the burden for implementing meta-DPPs is high: they have to do data reorganization for parallel execution in accordance with a data model and also provide infrastructure for parallel execution. That said, meta-DPPs can be re-used many times. In particular, VTK-m has only five meta-DPPs, but its algorithms contain over eighty instances of using meta-DPPs. Further, this usage will grow as more visualization algorithms are developed.

The remainder of this section describes our five proposed meta-DPPs: Visit Point With Cells (3.2.1), Visit Cell With Points (3.2.2), Point Neighborhood (3.2.3), Reduce By Key (3.2.4), and Map Field (3.2.5). Further, Table 1 provides a high-level comparison of these five meta-DPPs. The section then concludes with a discussion of modifiers for scatter and mask patterns (3.2.6). Finally, implementation details of the meta-DPPs can be found in our supplemental material.

3.2.1. Visit Point With Cells

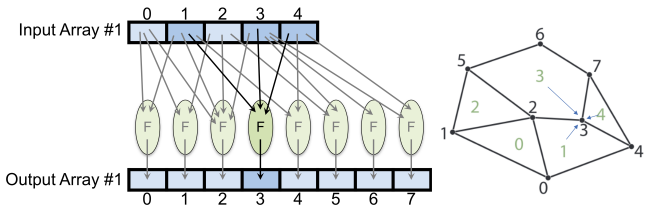

Figure 2: Data access pattern for Visit Point With Cells. This example takes a single cell-based array and generates a single point-based array. The example data set has eight points, so eight instances of the functor (F) are needed, one for each point. The input to each functor invocation are the incident cells, e.g., point 3 receives cell data from its incident cells (1, 3, and 4).

VisIt Point With Cells parallelizes over the point elements in a mesh. One of its inputs is the set of cells defining the input

Meta-DPP	Parallelizes Over	Inputs and Outputs	Data Available to Functor
Visit Point With Cells	Point elements in a mesh	(CF*, PF* → CF+)	Cells incident to a point
Visit Cell With Points	Cell elements in a mesh	(CF*, PF* → PF+)	Points incident to a cell
Point Neighborhood	Point elements in a mesh	(CF*, PF* → PF+)	Neighborhoods around a point
Reduce by Key	Unique values in a set of keys	(KA, VA*, → OA+)	Array values with same key
Map Field	Elements in field array	(CF* → CF+) or (PF* → PF+)	Array values at same element

Table 1: Comparing the five Meta-DPPs with respect to what they parallelize over, their inputs and outputs, and the data available to a functor. The symbol "CF*" means zero or more fields defined on cells, while "PF*" means zero or more fields defined on points. The "+" variants indicate one or more fields and are used for the outputs, since each meta-DPP produces at least one array. Putting it all together, the Visit Point With Cells Meta-DPP takes zero or more cell fields and point fields as input, and produces one or more cell fields as output. For Reduce by Key, "KA" refers to an array of keys, "VA*" refers to zero or more arrays of values for the input, and "OA+" refers to one or more arrays of values for the output. The number of elements for KA and each VA must be the same, while the number of elements in each OA will always be fewer (unless every key is unique).

domain (i.e., the points being visited). It also can accept any number of fields on the cells and points as input. Its output is one or more field arrays, sized to the number of points in the mesh.

For each point of a mesh, this meta-DPP retrieves information about the cells that are incident on that point, as demonstrated in Figure 2. Any input fields are also collected and passed to the functor provided to the meta-DPP.

In terms of utility for visualization algorithm developers, this operation is often used to interpolate cell-based field values to vertices. Canonical use cases include recentering and generating vertex normals.

3.2.2. Visit Cell With Points

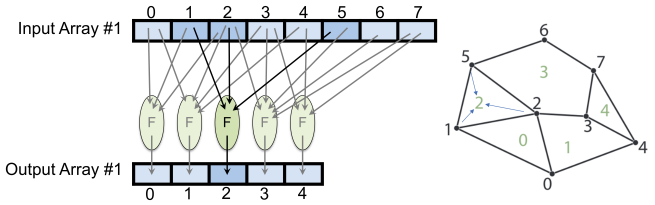

Figure 3: Data access pattern for Visit Cell With Points. This example takes a single point-based array and generates a single cell-based array. The example data set has five cells, so five instances of the functor (F) are needed, one for each cell. The input to each functor invocation are the incident points, e.g., cell 2 receives point data from its incident points (1, 2 and 5).

Visit Cell With Points parallelizes over the cell elements in a mesh. It has the same inputs and outputs and Visit Point With Cells: one of its inputs is the set of cells defining the input domain to parallelize over (i.e., the cells being visited), and it can also accept any number of fields on the cells or points. Once again, the output is one or more field arrays, but for this meta-DPP, the arrays are sized to the number of cells in the mesh.

For each cell of a mesh, this meta-DPP retrieves information about the shape of the cell and the incident points for that cell, as demonstrated in Figure 3. The location of the points as well as any input fields are also collected and passed to the functor provided to the meta-DPP.

In terms of utility for visualization algorithm developers, this operation is often used to operate on sampled point fields as continuous fields. It is also used to perform transformations of topology. Canonical use cases include operations such as gradient calculation and identifying cases for Marching Cubes [37].

3.2.3. Point Neighborhood

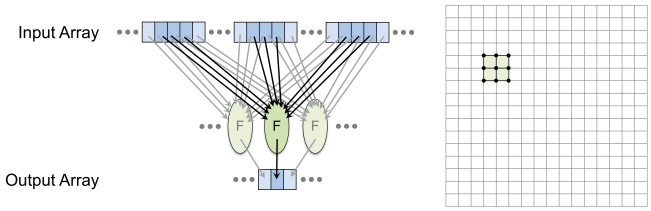

Figure 4: Data access pattern for Point Neighborhood. This example takes 2D structured cells and requests a point neighborhood with a "radius" of size 1. The radius determines neighbors in every direction resulting in a 3×3 patch of points. The input to each functor invocation are from the points within this neighborhood. To reduce the complexity of this figure, the inputs for only one example functor are shown, as well as the functors on the immediate "left" and "right" for reference. This meta-DPP is also capable of looking at larger neighborhoods, i.e., two or more cells away, and of course 3D meshes are supported.

Point Neighborhood parallelizes over the point elements in a mesh, and only works on meshes composed of *structured* cells. Its inputs include the set of cells and any number of fields defined on the points, and its output is one or more field arrays sized to the number of points in the mesh.

As demonstrated in Figure 4, for each point P in the mesh, this meta-DPP retrieves information about P, all points A_i adjacent to P, all points adjacent to A_i, and so on. The number of hops is determined by a user-specified neighborhood size.

In terms of utility, this operation is often used to filter point data by applying a kernel of the neighborhood size. Canonical use cases include denoising data and estimating derivatives with finite differences.

3.2.4. Reduce By Key

Reduce By Key parallelizes over unique values in a set of keys. Its input domain is defined by a "key" array, and it also accepts as input any number of "value" arrays. All the input arrays (1 key array plus n value arrays) must be the same length. The output is one or more arrays containing an entry for each *unique* key in the "key" array. The size of the output can be (and almost always is) smaller than the input.

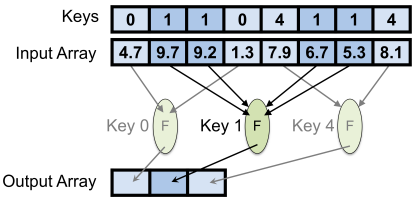

Figure 5: Data access pattern for Reduce By Key. This example takes one array for the keys and another array of data. The meta-DPP identifies that there are three unique keys (0, 1, and 4) and so invokes three functors (one for each key). The input to each functor are the elements of the array that match its key, e.g., the middle functor (which corresponds to key 1) receives input values 9.7, 9.2, 6.7, and 5.3.

Before executing, the Reduce by Key meta-DPP finds all duplicated keys in the "key" array. As demonstrated in Figure 5, it then collects all the values associated with a key and passes an array of these values to the functor provided to the meta-DPP.

In terms of utility, Reduce By Key is a powerful tool for combining mesh elements that are nearby or coincident. It can be used to find coincident points generated by parallel threads in algorithms like Marching Cubes and mesh subdivision by using edges of the original mesh as the keys [38]. It can also be used to find nearby points by using keys associated with bins in space [39].

It should be noted that the Reduce By Key meta-DPP is similar in concept to the ReduceByKey DPP. The biggest operational difference between the two is the functor passed to each. The ReduceByKey DPP requires a binary operation that reduces two items to a single item, and this binary operation is applied recursively to all items in the array. In contrast, the Reduce By Key meta-DPP takes a functor that accepts all values to reduce at once. This allows the meta-DPP to implement more complex operations not possible with the DPP version. For example, using spatial bins to combine nearby points (as described in [39]) requires a re-comparison of all items with the same key. This is trivial with the Reduce By Key meta-DPP but not directly possible with the simple binary operator of the DPP version.

3.2.5. Map Field

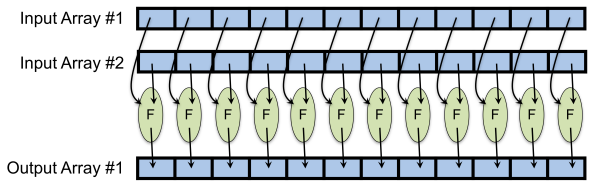

Figure 6: Data access pattern for Map Field. This example takes two input arrays and generates an output array. The example arrays each have twelve elements, so twelve instances of the functor (F) are needed, one for each element. The input to each functor invocation are the corresponding elements from the input arrays, i.e., the i^{th} functor receives the i^{th} elements from input arrays #1 and #2.

Map Field parallelizes over the elements in a field array. This meta-DPP can take any number of arrays as input and any number of arrays as output. Further, an array can also act as both an input and an output. That said, all arrays are expected to be the same size.

Conceptually, the operation of Map Field is quite simple. This meta-DPP calls the functor once for every element in the array as shown in Figure 6. Values from the appropriate index are pulled from the arrays and provided to the functor rather than giving the functor direct access to the arrays.

In terms of utility, Map Field is used to implement mathematical expressions on fields, for example finding the magnitude of a vector field. Further, it is often useful for scheduling parallel algorithms that are not well covered by the provided DPPs and meta-DPPs.

3.2.6. Modifiers (Scatter and Mask)

As described earlier, one of the typical challenges of implementing parallel visualization algorithms is the non-uniform access to inputs and outputs. It is often the case that the input data get sub-selected based on those that match certain conditions. Likewise, it is common that each input datum produces a different sized output. For example, in the famous Marching Cubes algorithm [37], many of the input cells that do not contain the isosurface can be ignored, and each cell that does contain the isosurface can generate anywhere between 1 and 6 polygons.

This jagged input and output is extremely problematic for the base DPP algorithms as it breaks the typical 1:1 correspondence between input and output that they assume. Our meta-DPPs handle jagged input and output by applying modifiers to their execution. These modifiers change the indexing of the meta-DPP's inputs and outputs in arbitrary patterns to match the algorithm being implemented.

The first type of modifier is called a scatter. A scatter can be used to define how many output items each input item creates. For example, a scatter can define a meta-DPP to generate 2 or more outputs for each input, which will increase the number of parallel operations executed. A scatter can also define a meta-DPP to skip over an input by assigning 0 outputs for that input. In general, these concepts can be mixed by assigning a count to each input specifying how many outputs to generate as demonstrated in Figure 7.

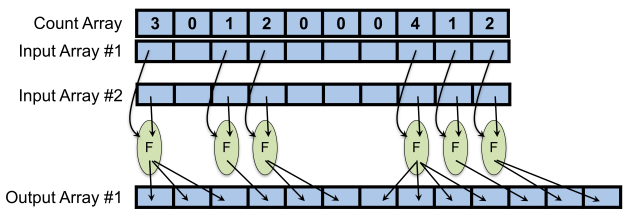

Figure 7: Data access pattern for the Scatter modifier. This example takes three arrays: two regular arrays (input arrays #1 and #2) and one that specifies the desired number of executions for each element of the arrays. In this example, the 0^{th} element of the "count array" has value three, so the specified functor F will be executed three times, each time taking the 0^{th} element of the input arrays as input. The next element has value zero, so no functors will be executed operating for those array elements. Finally, the total size of the output array is the sum of the count array.

Figure 7 shows a scatter applied to a simple Field Map meta-DPP, but the scatter modifier can be applied to any type of meta-DPP. For example, a scatter can be applied to a Visit Cell with Points meta-DPP to select which cells to apply an operation to and how much data to generate. Managing the multiple levels of indexing required for an operation of this nature is complex, but VTK-m handles this internally. Algorithm creators can focus on the operations specific to the unit of operation.

The second type of modifier is called a mask. A mask can be used to define which items in the output to generate and which items to skip over. This behavior is similar to a scatter that uses counts of 0 and 1. However, the mask allows the meta-DPP to place results in an existing array of a larger size.

For example, a meta-DPP might be constructed to take an array of particle positions and advance them in a vector field. This meta-DPP could be iteratively invoked to trace the path of these particles. As the particles advance, some of the trajectories will naturally terminate because they hit a sink or left the domain. This leaves the algorithm with an array of particles, some of which should be advanced and others that do not move. A mask modifier on the meta-DPP allows the algorithm to sub-select the particles that are still active. This will schedule the meta-DPP only on those active particles, which makes the execution efficient without having to reshuffle the particles.

3.3. Convenience Routines

As with any library, VTK-m provides numerous convenience routines, which are simple to use features that perform a common operation. Unlike meta-DPPs, which are designed to be configurable and general, convenience routines have a fixed interface and perform a specific function. This makes convenience routines very easy to use but of limited applicability. The convenience routine encapsulates the more complicated use of DPPs or meta-DPPs.

We do not attempt to list every reusable piece of software available inside VTK-m. Instead, we limit our discussion to those convenience routines that satisfy the following two criteria. (1) The implementation of the convenience routine uses a DPP or meta-DPP in its implementation, thus simplifying per the developer efficiency metrics used in Section 4.1. (2) The convenience routine is used at least once by the algorithms analyzed in Section 4.1.

ArrayRange Given an array, ArrayRange finds the minimum and maximum value in that array. If the array contains vectors (such as point coordinates), then ArrayRange will find the minimum and maximum of each component.

CountToOffset Because visualization algorithms often deal with jagged data, it is common to need to pack items of different sizes into a larger array. Often an algorithm will start with a count of how many components are with each group (e.g. a count of how many vertices are in each cell of an unstructured grid such as the first has 8 vertices, the second has 4 vertices, etc.). To pack this data in an array, you really need the offsets of where each group starts (e.g. the first cell starts at index 0 of the connections array, the second at index 8, the third at index 12, etc.). CountToOffset will efficiently compute the necessary offsets from the counts.

Locators An algorithm sometimes needs to identify which cell in a mesh contains a point at a given coordinate. For irregular meshes, finding these cells efficiently requires special search structures. VTK-m comes with several types of cell locators, but the only one that requires DPPs or meta-DPPs to generate is a locator based on a two-level grid [40].

MapFieldMergeAverage Visualization algorithms sometimes need to merge elements together. Often this is a simplification of a mesh with elements that are coincident or that can be combined with minimal error. When elements are merged, the fields on the elements need to be combined in some way. A straightforward and often valid combination is to average the field values. MapFieldMergeAverage performs this averaging by reusing the data structures of a previous call to the Reduce By Key meta-DPP.

MapFieldPermutation Many visualization algorithms modify the structure of a mesh and need to pass data according to the modifications. For example, a threshold algorithm will remove cells from the mesh. MapFieldPermutation can reorder the cell fields on the input mesh to match the new cell ordering of the output mesh.

3.4. D^3: Data Management, DPPs, and Devices

The "D^3" portion of our design leverages three existing constructs from previous systems. The remainder of this section discusses relevant aspects of these constructs for MCD^3.

3.4.1. Data Management

With our MCD^3 design, we make the data management layer responsible for collecting the necessary data for a meta-DPP's functor or convenience routine to execute. This design is inspired by previous work in both visualization libraries like Dax [28] and EAVL [29], and general-purpose environments like that provided by ispc [41]. The data management layer operates by decoupling memory layout from execution — data is pulled from arrays and delivered to functors as arguments prior to their execution. This approach has two big benefits. First, it insulates meta-DPP and convenience routines — and by extension visualization algorithm developers — from data layout and reorganization issues. Second, it enables a single code base to support many data layouts. From the perspective of the functor, the input data is always points, cells, etc., and this data is always organized in the same way. This is possible because the data management layer performs the work of reorganizing the data prior to functor execution. With our design, the reorganization can manage both standard data layout issues (e.g., structure-of-arrays versus array-of-structures) and data model issues (e.g., isolating cells from unstructured meshes or regular meshes). These reorganizations are non-trivial, but they are done in the core MCD^3 design, meaning that visualization algorithm developers are spared.

The data management layer is crucial for our overall goals. In situ processing is a major motivator for our design, and simulation codes have varied data layouts and data models. Through the data management layer, MCD^3 is able to adapt without penalty.

3.4.2. Data-Parallel Primitives (DPPs)

Section 2.2 provided details on DPPs. For our design, we found the following DPPs to be useful: Copy, CopyIf, CopySubRange, CountSetBits, Fill, LowerBounds, Reduce, ReduceByKey, ScanInclusive, ScanInclusiveByKey, ScanExclusive, ScanExclusiveByKey, ScanExtended, Sort, SortByKey, Synchronize, Transform, Unique, UpperBounds. Further description of the DPPs can be found in our supplemental material.

3.4.3. Devices

The device represents the physical hardware on which a MCD^3 algorithm is run. Different devices from different vendors require different compilers with different language syntax to generate programs. Our MCD^3 approach assumes that the compiler for each device provides a common base language (C++ in the case of our implementation), and the extended language features are hidden from developers using MCD^3.

The data management and DPPs discussed previously work together to unify the interface to devices and simplify porting. The data management, in addition to providing a common interface to varying layouts, prepares data for use on devices. This could include copying data for devices with separate memory spaces or allocating memory for managed uniform memory.

Each device has a unique implementation for the DPPs. Because each DPP can be customized for each device, the MCD^3 design can provide the best possible implementation for each device. The DPP implementations work with the data management to place data in the appropriate location and format for the device.

4. Results

We organize results into three areas:

- Section 4.1 evaluates developer efficiency.
- Section 4.2 considers the efficacy of the overall MCD^3 system.
- Section 4.3 evaluates performance efficiency.

The results in the first two areas consider the usage of MCD^3 constructs within the VTK-m library. To generate the data for these results, we analyzed the current collection of 57 algorithms within the VTK-m source code. The process for collecting this data was automated, and is discussed in Appendix A. Finally, several figures in this section refer to VTK-m's algorithms by name. While the names are mostly self-explanatory, description of each algorithm can be found in the VTK-m's User Guide [42].

4.1. Evaluating Developer Efficiency

This section focuses on developer efficiency. As discussed in Section 3, the MCD^3 design benefits from three layers of efficiency: (1) meta-DPPs and convenience routines taking the place of multiple DPPs, (2) the data management layer providing support for many data layouts and mesh types, and (3) the DPP and device layers providing support for multiple hardware architectures. The benefits of the latter two layers have already been established [4, 29, 30, 43], so our focus is on evaluating the benefit of meta-DPPs and convenience routines.

Figure 8 shows the rate at which our meta-DPPs and convenience routines are used in VTK-m. The rates inform the opportunity for benefit, i.e., visualization algorithm developers can only benefit if these constructs are used at a high rate. Further, this usage is indeed high — the 57 visualization algorithms use a total of 85 meta-DPPs, 44 convenience routines, 32 Copy DPPs, and 78 non-Copy DPPs. (The Copy DPP is differentiated since it is so fundamental that developers may not even be aware that they are even using a DPP.) Of all the algorithms, 97% (all but two) use at least one meta-DPP, 47% use a convenience routine, 56% use a Copy DPP and 47% use a non-Copy DPP. On average, an algorithm uses 1.5 meta-DPPs, 0.8 convenience routines, 0.6 Copy DPPs and 1.4 non-Copy DPPs.

Evaluating developer savings from meta-DPPs and convenience routines is difficult, and our process for doing so is an approximation. We started by studying each meta-DPP and convenience routine and analyzed how many DPPs each used. The results of this analysis are in Table 2.

	Feature	# DPPs
meta-DPP	Visit Point With Cells	6
	Reduce By Key	6
	Scatter Counting	4
	Visit Cell With Points	1
	Point Neighborhood	1
	Map Field	1
Convenience	Locator	18
	MapFieldMergeAverage	6
	FieldMapPermutation	1
	CountToOffset	1
	ArrayRange	1

Table 2: Number of DPPs used for each meta-DPP and convenience routine.

We then combined this analysis with our data from Figure 8 to estimate developer cost with and without MCD^3. Formally, for a visualization algorithm $\mathcal{A}$ and a programming construct X (where $X \in$ {meta-DPP, DPP, convenience}), we define the following:

- $Uses(\mathcal{A}, X)$ is 1 if algorithm $\mathcal{A}$ uses X, 0 if it does not. Considering an example, if X is the Map Field meta-DPP and $\mathcal{A}$ uses Map Field three times, then $Uses(\mathcal{A}, X) = 1$.

- $Equiv(X)$ is the cost to implement X in a DPP-only system. $Equiv$ is the sum of two terms: (1) the number of

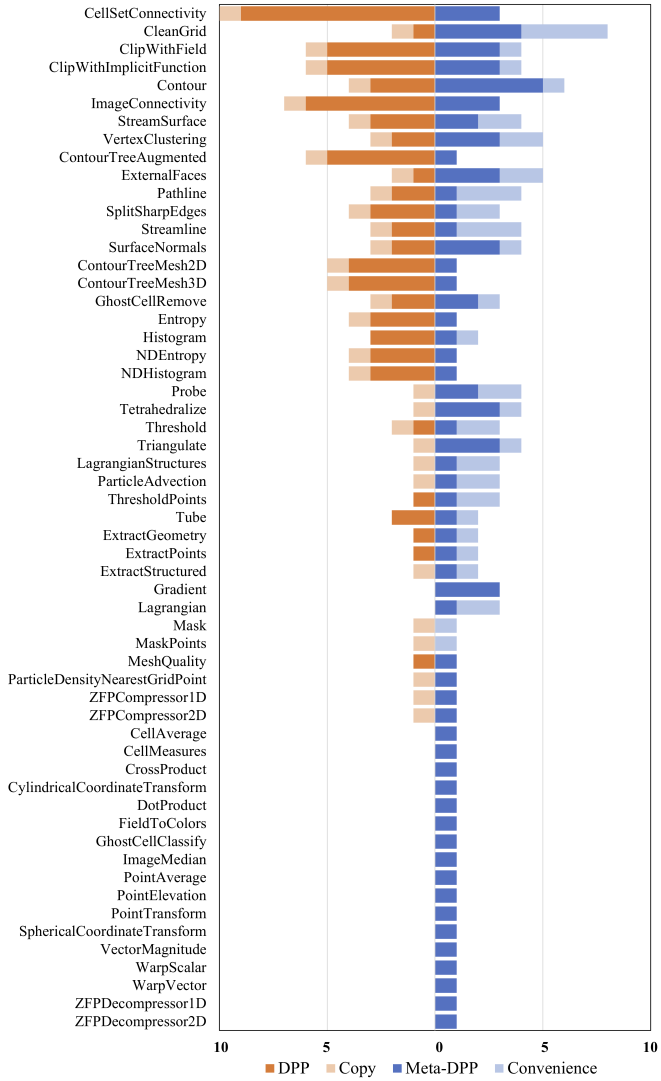

Figure 8: Counting the usage of each MCD^3 construct by visualization algorithm. The Copy DPP is separated from the other DPPs due to its trivial nature (literally copying an array), and since visualization algorithm developers often do not even realize they are using a DPP when they invoke it.

DPPs required to implement X (taken from Table 2) and (2) the data layout and management cost. For the data layout and management cost (which can be substantial), we use a flat value of 1, i.e., a savings equivalent to calling one DPP. We believe this is a reasonable floor to the complexity of an efficient data management layer. Again considering the example of Map Field, we would assign $Equiv(MapField) = 2$, since Table 2 has a score of 1 and the data management for Map Field (arranging many arrays, designating what is an input and what is an output, etc.) receives the flat value of 1.

Using the functions *Uses* and *Equiv*, we can then define functions for calculating algorithm development costs. We define the following: $C_{MCD^3}(\mathcal{A})$ (cost estimate for developing algorithm $\mathcal{A}$ using MCD^3), $C_{DPP-Only}(\mathcal{A})$ (cost estimate using only DPPs): and $C_{DPP+Conv}(\mathcal{A})$ (cost estimate for developing algorithm $\mathcal{A}$ using only DPPs and convenience routines). This final cost, $C_{DPP+Conv}(\mathcal{A})$, acknowledges the fact that convenience

Algorithm Uses	Savings Factor DPP-Only	DPP + Conv
Visit Points with Cells	4.1X	3.5X
Visit Cells with Points	3.2X	2.3X
All Other	2.7X	1.4X
Other: Particle Advection	8.8X	1.4X
Other: Non-Particle Advection	1.6X	1.4X

Table 3: Development cost savings factor of MCD^3 compared to DPP-Only and DPP+Convenience using data shown in Figure 9. These savings factors are computed as $C_{DPP-Only}/C_{MCD^3}$ and $C_{DPP+Conv}/C_{MCD^3}$, respectively. The cost savings factors are broken up into three major categories: algorithms that use Visit Points with Cells MCD^3, algorithms that use Visit Cells with Points MCD^3, and all others. Because of the large cost differential between algorithms in the Other category that use particle advection and those that do not, it is broken out into separate categories.

routines could be used in a DPP-only system. Formally, these definitions are as follows:

$$C_{MCD^3}(\mathcal{A}) = \sum_{m \in M} Uses(\mathcal{A}, m) + \sum_{d \in D} Uses(\mathcal{A}, d)$$

$$C_{DPP-Only}(\mathcal{A}) = \sum_{m \in M} Uses(\mathcal{A}, m) \times Equiv(m) + \sum_{c \in C} Uses(\mathcal{A}, c) \times Equiv(c) + \sum_{d \in D} Uses(\mathcal{A}, d)$$

$$C_{DPP+Conv}(\mathcal{A}) = \sum_{m \in M} Uses(\mathcal{A}, m) \times Equiv(m) + \sum_{d \in D} Uses(\mathcal{A}, d)$$

where M is the set of all meta-DPPs, C is the set of all convenience routines, and D is the set of all DPPs. Of note, C_{MCD^3} and $C_{DPP+Conv}$ do not include a cost for convenience routines, since they can be invoked with a single line of code, and also $C_{DPP-Only}$ does not apply the *Equiv* function to DPPs, since our counting is based on the number of DPPs (i.e., $Equiv(d) = 1$ for any DPP d).

Figure 9 shows the results of this analysis for the 57 visualization algorithms in our corpus. The total implementation cost for these algorithms in MCD^3 is 195 (i.e., $\sum_{a \in VisAlgs} C_{MCD^3}(a)$), while the total implementation cost in a DPP-only environment is 605 (i.e., $\sum_{a \in VisAlgs} C_{DPP-Only}(a)$), for a savings of 3.1X in the aggregate. The total implementation cost in a DPP-only plus convenience routines environment is 399 (i.e., $\sum_{a \in VisAlgs} C_{DPP+Conv}(a)$), for a savings of 1.5X in the aggregate. The data in Table 3 further quantifies the implementation savings using MCD^3 by the algorithm categories used in Figure 9 (i.e., blue bars for Visit Point with Cells, red bars for Visit Cells with Points, and orange bars for Other). Within the Other algorithms, those that involved particle advection had significant savings compared to the $C_{DPP-Only}$ implementation due to the locator convenience routines. The Other algorithms that did not involve particle advection were generally simple algorithms doing one or two operations and likely would not justify the MCD^3 design on their own.

Overall, however, we feel the savings are significant for visualization algorithm developers, and the magnitude of savings across all visualization algorithms justifies the cost of developing the MCD^3 system. For this latter point, the investment in the system can continue to be amortized in the future, since additional algorithms will incur no further MCD^3 development costs.

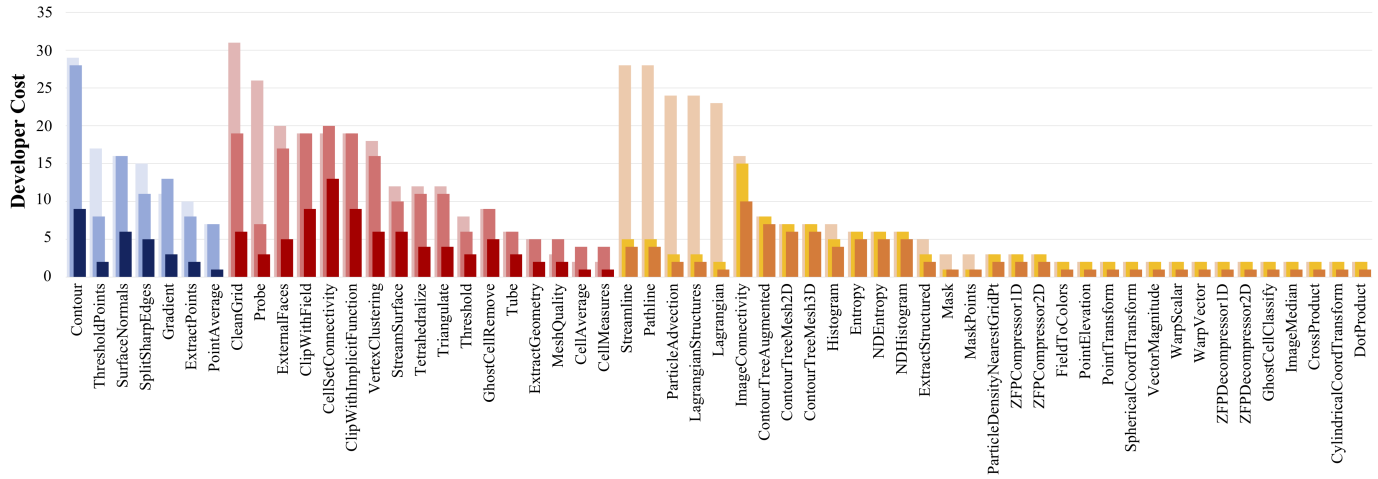

Figure 9: Comparing development costs for each of the algorithms in VTK-m using our C_{MCD^3}, $C_{DPP+Conv}$ and $C_{DPP-Only}$ functions. The algorithms are grouped into three categories: those that use only the Visit Point with Cells meta-DPP (colored blue), those that use only the Visit Cell with Pointss meta-DPP (colored red), and all other algorithms (colored orange). Each algorithm has three vertical bars: the darkest color is C_{MCD^3}, the medium color is $C_{DPP+Conv}$, and the lightest color is $C_{DPP-Only}$.

4.2. System Efficacy

With this section, we seek to evaluate the efficacy of our choices for our constructs (i.e., meta-DPPs, convenience routines, and DPPs) and their specific instances (e.g., Visit Point With Cell, Array Range, and CopyIf). Our choices could be poor if we hit either extreme with respect to complexity. First, it could be overly complicated, i.e., we have generated more constructs/instances than are needed to solve our target problem. Second, it could be too simple, i.e., we do not have sufficient constructs/instances to solve our target problem.

Is the MCD^3 design overly complicated? We feel the answer to this question is no, since all of our constructs (meta-DPP, convenience routine, and DPP) are used, and many of the instances are heavily used (see Figure 10 for details). While DPPs represent half of all usage, the other constructs represent the other half, justifying their inclusion. Of course, this analysis is not definitive, as some instances could potentially be removed, and their functionality could be replaced by a combination of others.

Is the MCD^3 design too simple? We feel the answer to this question is again no, since the design is expressive enough to implement diverse visualization algorithms, and more importantly the visualization algorithms we find to be useful. We offer two pieces of evidence that the algorithms we have implemented provide a useful feature set for stakeholders. First, VTK-m is the main visualization library behind Ascent [44], an in situ visualization library that is used by over a dozen computational simulation teams. That VTK-m can meet these teams' collective needs is significant evidence that its 57 algorithms provide a sufficient feature set. Second, at the inception of the VTK-m project, the team identified 20 core algorithms that it felt were necessary to deliver a useful product. Each of these algorithms is now implemented in VTK-m, and some required more exotic usage than originally envisioned, like the use of hash tables. Of course, these two pieces of evidence do not exhaustively cover the space of possible visualization algorithms,

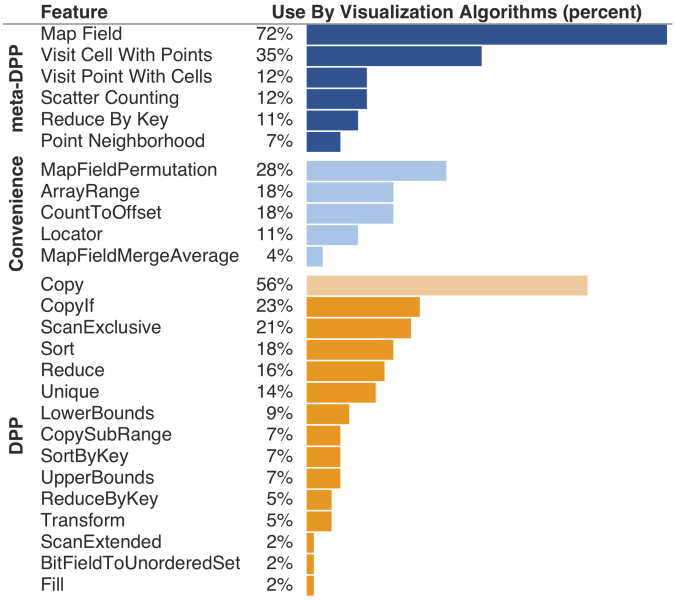

Figure 10: Measuring the usage by visualization algorithms of individual meta-DPPs (dark blue), convenience routines (light blue), and DPPs (orange). Each row corresponds to usage for a given operation. For example, the MapField meta-DPP was used by 72% of the visualization algorithms (41 of the 57). (Note that any DPP not used directly by any visualization algorithm implementation are not listed here. Hence, some DPPs listed in Section 3.4.2 used elsewhere in VTK-m are not listed here.)

and it is possible that a future algorithm will be difficult to fit within the framework.

4.3. Evaluating Performance Efficiency

To evaluate MCD^3 performance, we surveyed ten studies on MCD^3, and collate their results. We feel this provides a holistic picture of MCD^3 performance, as it evaluates a considerable number of algorithms, platforms, data sets, and comparators. That said, it is not possible to directly link these studies with

individual algorithms from 4.1, since some studies consider MCD^3-based modules that are not considered "algorithms" (filters) in VTK-m (like hashing and ray tracing) and others consider combined performance of multiple algorithms. Regardless, we feel the existing set of published performance studies greatly informs the performance of all MCD^3 algorithms.

We divide our results into two sections. The first section (4.3.1) evaluates how the hardware-agnostic approach of MCD^3 performs versus hardware-specific comparators for nine different algorithms. The second section (4.3.2) considers studies that focused on scaling properties on CPU hardware — when given more cores, did performance improve proportionally? This section considers five such studies, although four of the studies also had hardware-specific comparators and appeared in section 4.3.1.

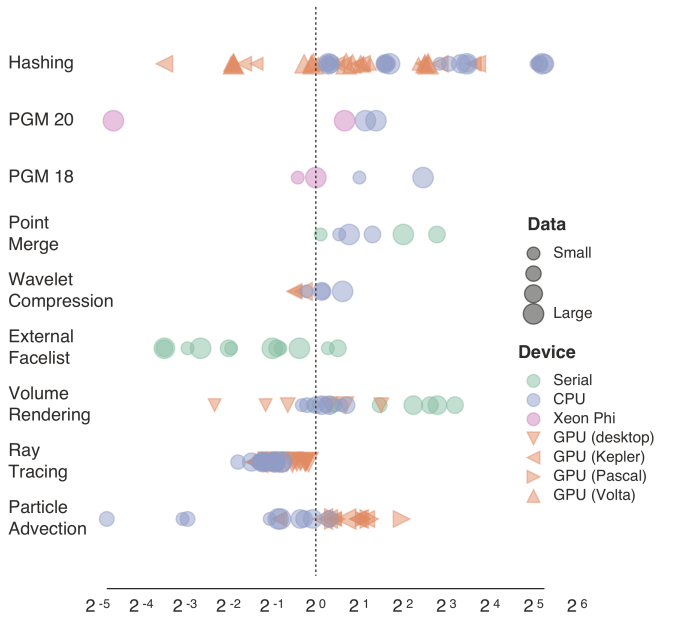

Figure 11: MCD^3 performance against hardware-specific comparators for nine algorithms. The horizontal axis represents the ratio in MCD^3 performance against its comparator — 2^{-1} means MCD^3 took twice as long, 2^1 means MCD^3 took half the time, and 2^0 (1) means MCD^3 took the same amount of time. The glyphs indicate both the hardware type (glyph color and glyph type) and the size of the data set being processed (glyph size). For data size, all plotting is relative — the largest data set considered in that study gets the largest glyph size, the smallest data set gets the smallest glyph size, and the other glyphs are scaled proportionally between the two extremes. Finally, specifics about hardware and comparators can be found in Table 4.

4.3.1. MCD^3 Versus Hardware-Specific Comparators

Table 4 and Figure 11 show the results from nine studies comparing individual algorithms. The hardware-specific comparators came from a mix of well-known visualization/rendering software (Embree [53], HAVS [54], OptiX [55], VisIt [2], VTK [8], Vapor [56]) and direct implementations (CUDA, OpenMP, pthreads, TBB, Thrust). While Table 4 lists the range of comparisons, Figure 11 shows the outcome for each individual experiment.

Algorithm	Architecture	Comp.	Perf.
Ray Tracing [45]	I I7 4770K	Embree	0.28-0.48
	I Ivy Bridge	Embree	0.4-0.58
	N GTX Titan Blk	OptiX	0.44-0.56
	N Tesla K80M	OptiX	0.37-0.51
	N GeForce 750Ti	OptiX	0.69-0.89
	N GeForce 620M	OptiX	0.73-1.16
Volume Rendering [46]	I Ivy Bridge (1)	VisIt	0.73-9.1
	I I7 4770K (8)	VTK	2
	N GTX Titan Blk	HAVS	0.33-2
External facelist [47]	I Ivy Bridge (1)	VTK	0.50-1.4
	I Ivy Bridge (1)	VisIt	0.08-0.25
Wavelet Compression [48]	I Haswell (16)	Vapor	0.8-1.5
	N Tesla K40	CUDA	0.6-0.8
Particle Advection [49]	I Ivy Bridge (16)	VisIt	2.4-3.6
	I Haswell (28)	pthreads	0.03-1.6
	IB Power8 (20)	pthreads	0.05-1.03
	N Tesla K20x	CUDA	0.37-2.26
	N Tesla K80	CUDA	0.54-2.45
	N Tesla P100	CUDA	0.48-4.08
Point Merge [39]	IB Power9 (1)	VTK	1.07-6.86
	IB Power9 (40)	VTK	1.4-2.5
	N Tesla V100	VTK-m	0.48-4.0
Probabalistic Graphical Modeling (PGM) 2018 [50]			
	I Ivy Bridge (24)	OpenMP	2-7
	I Phi 7250 (68)	OpenMP	0.75-4.25
PGM 2020 [51]	I Ivy Bridge (8)	{OpenMP,	2.2, 2.6
	I Xeon Phi 7250	pthreads}	0.04, 1.58
Hashing [52]	I Skylake (32)	{TBB,	1.2-37
	N Tesla K40	CUDPP,	0.09-13
	N Tesla V100	Thrust}	0.27-5.96

Table 4: Studies comparing performance for algorithms implemented in MCD^3 against hardware-specific implementations. In the architecture column, numbers in parentheses specify number of multi-core CPU cores used in the experiments. In the comparator column (abbreviated as "Comp"), the last two studies picked the best from multiple comparators, and the set of comparators are listed with curly braces. Finally, in the performance column, the numbers represent the range of outcomes. As an example of how to interpret this table, the entry in the top row indicates that an MCD^3 ray tracing algorithm only had 28% to 48% of the performance (i.e., from almost 4X slower to nearly 2X slower) compared to Intel's Embree ray tracer when run on the Intel I7 architecture.

Two of the algorithms, external facelist and ray tracing, had significantly worse performance with MCD^3 implementations than with its hardware-specific comparators. For external facelist, the cause was an asymmetric comparison — the comparator was a serial algorithm that could skip any overhead necessary for any parallel algorithm. Specifically, the serial comparator could update its internal tables without fear of collisions, whereas the MCD^3 code had to be designed to prevent collisions when running in parallel. For ray tracing, the cause was that the comparators are extremely efficient — NVIDIA's

OptiX and Intel's Embree are the product of dedicated teams embedded at their respective hardware vendors. The difference in performance, then, is likely more a reflection of respective development time than a statement about MCD^3. Further, the MCD^3 approach actually beat OptiX on older NVIDIA cards, speaking to the extent that OptiX is tuned for the latest NVIDIA hardware and to the portable performance of MCD^3.

Algorithm	CPUs	GPUs	X. Phi	Serial	Total
External facelist	-	-	-	0.34	0.34
PGM 18	3.32	-	0.87	-	1.69
PGM 20	2.39	-	0.25	-	0.78
Particle advection	0.38	1.53	-	-	0.76
Point merge	1.82	-	-	3.10	2.38
Ray tracing	0.47	0.55	-	-	0.51
Volume rendering	1.13	0.83	-	3.10	1.43
Wavelet compression	1.13	0.75	-	-	0.92
Hashing	5.97	1.45	-	-	2.94
Total	1.45	0.95	0.47	1.48	**1.14**

Table 5: Aggregate performance of MCD^3 performance against hardware-specific comparators for nine algorithms on four hardware architectures. Each entry in the table represents a geometric mean over its experiments. For example, if algorithm X on hardware Y had three experiments, where MCD^3 was five times as fast, half as fast, and one fourth as fast as its hardware comparator, then the table will contain 0.85 (= $(5 \times 0.5 \times 0.25)^{\frac{1}{3}}$). We use the geometric mean since it captures aggregate behavior better than an arithmetic mean (which would be 1.92 for the previous example). We also combined results in algorithm and in hardware, i.e., if algorithms X, Y, and Z had results on hardware W, we calculated performance on W as the geometric mean of results for X, Y, and Z on W. One benefit of this approach is that is unaffected by the number of experiments run for a given study — if one study contained one hundred experiments and another study contained two, then the findings from the first study do not overshadow the second. Finally, we calculated the geometric mean over all hardware-algorithm pairs, which was 1.14.

Table 5 summarizes aggregate performance over the nine algorithms from their respective studies, and we use the methodology for calculating individual table entries as our best estimate at relative performance. Four of the nine algorithms are faster than their hardware comparators, whereas five are slower. This provides evidence that MCD^3 enables good performance, as the expected outcome if MCD^3 is as good as hardware-specific implementation would be a 50/50 mix of faster and slower. On the hardware side, we saw that serial and CPU experiments were faster. This could possibly demonstrate a benefit of DPP programming, as it is not possible to incorporate serial bottlenecks. We felt the GPU performance of 0.95X was quite good, and matches our team's experiences with good GPU performance. Similarly, we have felt that Xeon Phi performance was poor, which is borne out in the table. In particular, the study by Perciano et al. [51] demonstrated poor performance at high scale. MCD^3 performance was competitive at lower concurrency, but not when the hyperthreads were involved. This may indicate a shortcoming in Xeon Phi device adapters. Finally, the aggregation of the experiments demonstrates a 1.14X speedup from using MCD^3 over hardware-specific comparators. As previously discussed, these gains are coming from serial and multi-core CPU improvements, and likely from limiting programmers to only using fast programming constructs. That said, the GPU results (0.95X) indicate that MCD^3 is competitive.

4.3.2. MCD^3 Scaling on Multi-Core CPUs

For each of the studies that ran scaling studies on multi-core CPUs (not Xeon Phi), Table 6 shows the parallel efficiency and Figure 12 shows the behavior with increasing numbers of cores. For the most part, scaling is quite good, as the MCD^3 algorithms get proportionally faster as more cores are added. The two exceptions are for contour trees (where scalability is similar for a native OpenMP implementation due to algorithm complexities) and for the point merge algorithm when it switches to hyperthreading. That said, we find the overall scalability is good evidence that MCD^3 is an effective programming paradigm for multi-core CPUs.

Algorithm	Architecture	Max Cores	Parallel Efficiency
Volume Rendering [46]	I Ivy Bridge	24	0.73
External Facelist [47]	I Ivy Bridge	16	0.77
Contour Tree [57]	I Sandy Bridge	32	0.24
Point Merge [39]	IB Power 9	40	0.55
Particle Advection [49]	I Haswell	28	0.78

Table 6: Surveying multi-core CPU scaling studies for five visualization algorithms. While these studies considered many concurrency levels, this table shows the maximum concurrency, as well as the parallel efficiency achieved at that maximum concurrency.

5. Conclusion and Future Work

Overall, we feel our results demonstrate the efficacy of MCD^3 for our twin goals of minimizing developer time while achieving efficient portable performance on many-core architectures. For developer performance, we feel our analysis calculating the effort for a DPP-only equivalent library to VTK-m is compelling. While the 3.1X number has approximations, the

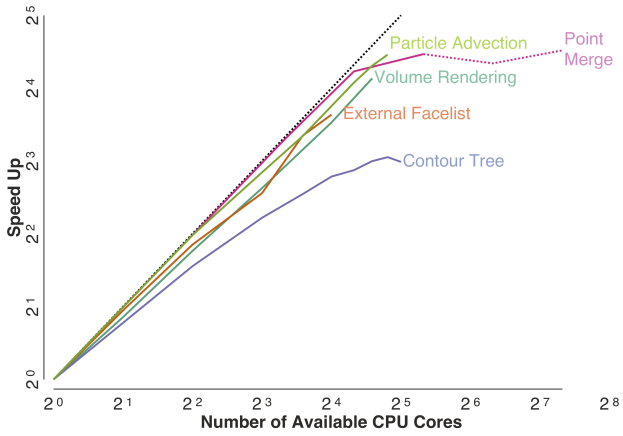

Figure 12: Plotting scaling study results for five visualization algorithms on multi-core CPUs. Each visualization algorithm is plotted as its own line, with a unique color, and the ideal scaling line is drawn as a dotted black line. Hyperthreading experiments for the Point Merge algorithm are indicated with a dotted line. These experiments go to 80 threads, since hyperthreading enabled running with double the number of physical cores.

number is high enough that we feel it clearly speaks to savings for the VTK-m development team due to MCD^3 principles, especially when combined with the usage of meta-DPPs and convenience routines. For efficient performance, we feel our analysis indicating that MCD^3 is faster than hardware-specific comparators (1.14X in total) is surprising. An ideal outcome would have been 1.0X, and — in our opinion — a realistic goal at the onset would have been to achieve a number between 0.8X and 0.9X. As discussed, we feel that part of our strong result is due to the disciplined nature of MCD^3 programming, which avoids serial bottlenecks. Regardless, the 0.95X result for GPUs is (in our opinion) compelling. Finally, the CPU scaling numbers provide further evidence of good performance overall.

There are additional benefits to MCD^3, in particular with meta-DPPs, that are not discussed in Section 4 because they are not measurable. For example, there are forms of scatters and masks that are not captured in Section 4 because they do not directly use a DPP but rather provide efficient and convenient index manipulation. Although these features are certainly valuable to developers, there is no way to identify in our corpus when an algorithm uses its own alternate index manipulation. To avoid such one-sided comparisons, we left these meta-DPP benefits out of Section 4.

There are several drawbacks for MCD^3. One is the time to implement the underlying system, although we feel this is amortized over saved developer time. Second, our MCD^3 implementation makes heavy use of template meta-programming, which raises the barrier to entry for developers and also increases compile time. Third, the MCD^3 design differs from a traditional approach, which also raises the barrier to entry — visualization algorithm developers must learn new constructs.

Future work centers around our twin goals of developer time and performance efficiency. We will continue to monitor the VTK-m library and see if new algorithms challenge the MCD^3 design (e.g., needing new meta-DPPs). We also will be interested to see performance studies on upcoming hardware, in particular Intel and AMD GPUs.

Acknowledgments

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

References

[1] J. Ahrens, B. Geveci, C. Law, ParaView: An End-User Tool for Large Data Visualization, The Visualization Handbook (2005) 162–170.

[2] H. Childs, et al., VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data, in: High Performance Visualization–Enabling Extreme-Scale Scientific Insight, 2012, pp. 357–372.

[3] G. E. Blelloch, Vector models for data-parallel computing, Vol. 356, MIT press Cambridge, 1990.

[4] K. Moreland, et al., VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures, IEEE Computer Graphics and Applications (CG&A) 36 (3) (2016) 48–58.

[5] E. W. Bethel, H. Childs, C. Hansen (Eds.), High Performance Visualization—Enabling Extreme-Scale Scientific Insight, CRC Press/Francis–Taylor Group, Boca Raton, FL, 2012.

[6] C. Upson, et al., The Application Visualization System: A Computational Environment for Scientific Visualization, Computer Graphics and Applications 9 (4) (1989) 30–42.

[7] G. Abram, L. A. Treinish, An extended data-flow architecture for data analysis and visualization, Research report RC 20001 (88338), IBM T. J. Watson Research Center, Yorktown Heights, NY, USA (Feb. 1995).

[8] W. J. Schroeder, K. M. Martin, W. E. Lorensen, The Design and Implementation of an Object-Oriented Toolkit for 3D Graphics and Visualization, in: Proceedings of the 7th conference on Visualization, IEEE Computer Society Press, 1996, pp. 93–ff.

[9] R. Frank, M. F. Krogh, The EnSight Visualization Application, in: High Performance Visualization-Enabling Extreme-Scale Scientific Insight, 2012, pp. 429–442.

[10] S. M. Legensky, Interactive Investigation of Fluid Mechanics Data Sets, in: Proceedings of the 1st conference on Visualization, IEEE Computer Society Press, 1990, pp. 435–439.

[11] S. Grottel, M. Krone, C. Müller, G. Reina, T. Ertl, Megamol-a prototyping framework for particle-based visualization, IEEE Transactions on Visualization and Computer Graphics 21 (2) (2015) 201–214.

[12] S. G. Parker, C. R. Johnson, Scirun: a scientific programming environment for computational steering, in: Proceedings of the ACM/IEEE Conference on Supercomputing, 1995, pp. 52–52.

[13] J. Clyne, P. Mininni, A. Norton, M. Rast, Interactive desktop analysis of high resolution simulations: application to turbulent plume dynamics and current sheet formation, New Journal of Physics 9 (8) (2007) 301.

[14] M. J. Turk, et al., yt: A Multi-code Analysis Toolkit for Astrophysical Simulation Data, The Astrophysical Journal Supplement Series 192 (Jan. 2011).

[15] A. C. Bauer, et al., In Situ Methods, Infrastructures, and Applications on High Performance Computing Platforms, Computer Graphics Forum 35 (3) (2016) 577–597.

[16] H. Childs, et al., A Terminology for In Situ Visualization and Analysis Systems, International Journal of High Performance Computing Applications 34 (6) (2020) 676–691.

[17] S. Ahern, et al., Scientific Discovery at the Exascale: Report for the DOE ASCR Workshop on Exascale Data Management, Analysis, and Visualization (July 2011).

[18] T. Peterka, et al., Priority research directions for in situ data management: Enabling scientific discovery from diverse data sources, The International Journal of High Performance Computing Applications 34 (4) (2020) 409–427.

[19] H. Childs, J. Bennett, C. Garth, B. Hentschel, In Situ Visualization for Computational Science, IEEE Computer Graphics and Applications 39 (6) (2019) 76–85.

[20] K. Gregory, A. Miller, C++ AMP: Accelerated Massive Parallelism With Microsoft Visual C++, Microsoft Press, 2012.

[21] S. Iwasaki, et al., BOLT: Optimizing OpenMP parallel regions with user-level threads, in: International Conference on Parallel Architectures and Compilation Techniques (PACT), 2019, pp. 29–42.
[22] J. Szuppe, Boost.Compute: A parallel computing library for C++ based on OpenCL, in: Proceedings of the 4th International Workshop on OpenCL, 2016, pp. 1–39.
[23] N. Bell, J. Hoberock, Thrust: A Productivity-Oriented Library for CUDA, in: GPU Computing Gems, Morgan Kaufmann, 2011, pp. 359–371.
[24] D. Weiskopf, GPU-based interactive visualization techniques, Springer, 2007.
[25] M. Ament, S. Frey, C. Müller, S. Grottel, T. Ertl, D. Weiskopf, GPU-Based Visualization, in: High Performance Visualization—Enabling Extreme-Scale Scientific Insight, 2012, pp. 223–260.
[26] M. B. Rodriguez, et al., A Survey of Compressed GPU-Based Direct Volume Rendering, in: Eurographics (STARs), 2013, pp. 117–136.
[27] J. Beyer, M. Hadwiger, H. Pfister, State-of-the-art in gpu-based large-scale volume visualization, in: Computer Graphics Forum, Vol. 34, Wiley Online Library, 2015, pp. 13–37.
[28] K. Moreland, U. Ayachit, B. Geveci, K.-L. Ma, Dax toolkit: A proposed framework for data analysis and visualization at extreme scale, in: IEEE Symposium on Large Data Analysis and Visualization, 2011, pp. 97–104.
[29] J. S. Meredith, S. Ahern, D. Pugmire, R. Sisneros, EAVL: the extreme-scale analysis and visualization library, in: Eurographics Symposium on Parallel Graphics and Visualization, 2012, pp. 21–30.
[30] L. Lo, et al., PISTON: A portable cross-platform framework for data-parallel visualization operators, in: Eurographics Symposium on Parallel Graphics and Visualization, 2012, pp. 11–20.
[31] P. S. McCormick, J. Inman, J. P. Ahrens, C. Hansen, G. Roth, Scout: A hardware-accelerated system for quantitatively driven visualization and analysis, in: IEEE Visualization, 2004, pp. 171–178.
[32] P. McCormick, et al., Scout: a data-parallel programming language for graphics processors, Parallel Computing 33 (10-11) (2007) 648–662.
[33] G. L. Kindlmann, et al., Diderot: a Domain-Specific Language for Portable Parallel Scientific Visualization and Image Analysis, IEEE Trans. Vis. Comput. Graph. 22 (1) (2016) 867–876.
[34] C. Chiw, et al., Diderot: a parallel DSL for image analysis and visualization, in: Proceedings of the SIGPLAN conference on Programming Language Design and Implementation, 2012, pp. 111–120.
[35] P. Rautek, S. Bruckner, M. E. Gröller, M. Hadwiger, ViSlang: A system for interpreted domain-specific languages for scientific visualization, IEEE Transactions on Visualization and Computer Graphics 20 (12) (2014) 2388–2396.
[36] H. Choi, et al., Vivaldi: A domain-specific language for volume processing and visualization on distributed heterogeneous systems, IEEE transactions on visualization and computer graphics 20 (12) (2014) 2407–2416.
[37] W. E. Lorensen, H. E. Cline, Marching Cubes: A High Resolution 3D Surface Construction Algorithm, Computer Graphics (Proceedings of SIGGRAPH 87) 21 (4) (1987) 163–169.
[38] R. Miller, K. Moreland, K.-L. Ma, Finely-Threaded History-Based Topology Computation, in: Eurographics Symposium on Parallel Graphics and Visualization, 2014, pp. 41–48.
[39] A. Yenpure, H. Childs, K. Moreland, Efficient point merging using data parallel techniques, in: Eurographics Symposium on Parallel Graphics and Visualization, 2019.
[40] J. Kalojanov, M. Billeter, P. Slusallek, Two-Level Grids for Ray Tracing on GPUs, Computer Graphics Forum 30 (2) (2011) 307–314.
[41] M. Pharr, W. R. Mark, ispc: A SPMD compiler for high-performance CPU programming, in: Innovative Parallel Computing, 2012, pp. 1–13.
[42] K. Moreland, The VTK-m User's Guide, version 1.5, Tech. Rep. SAND 2019-12638 B, Sandia National Laboratories (2019).
[43] K. Moreland, B. King, R. Maynard, K.-L. Ma, Flexible analysis software for emerging architectures, in: 2012 SC Companion (Petascale Data Analytics: Challenges and Opportunities), 2012, pp. 821–826.
[44] M. Larsen, et al., The ALPINE In Situ Infrastructure: Ascending from the Ashes of Strawman, in: Proceedings of the Workshop of In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization, Denver, CO, 2017, pp. 42–46.
[45] M. Larsen, et al., Ray Tracing Within a Data Parallel Framework, in: IEEE Pacific Visualization Symposium, 2015, pp. 279–286.
[46] M. Larsen, S. Labasan, P. A. Navrátil, J. S. Meredith, H. Childs, Volume Rendering Via Data-Parallel Primitives, in: EuroGraphics Symposium on Parallel Graphics and Visualization, 2015, pp. 53–62.
[47] B. Lessley, R. Binyahib, R. Maynard, H. Childs, External facelist calculation with data-parallel primitives., in: EuroGraphics Symposium on Parallel Graphics and Visualization, 2016, pp. 11–20.
[48] S. Li, et al., Achieving Portable Performance For Wavelet Compression Using Data Parallel Primitives, in: EuroGraphics Symposium on Parallel Graphics and Visualization, 2017, pp. 73–81.
[49] D. Pugmire, et al., Performance-Portable Particle Advection with VTK-m, in: EuroGraphics Symposium on Parallel Graphics and Visualization, 2018, pp. 45–55.
[50] B. Lessley, et al., DPP-PMRF: Rethinking Optimization for a Probabilistic Graphical Model Using Data-Parallel Primitives, in: IEEE Symposium on Large Data Analysis and Visualization, 2018, pp. 34–44.
[51] T. Perciano, C. Heinemann, D. Camp, B. Lessley, E. W. Bethel, Shared-Memory Parallel Probabilistic Graphical Modeling Optimization: Comparison of Threads, OpenMP, and Data-Parallel Primitives, in: International Conference on High Performance Computing, 2020, pp. 127–145.
[52] B. Lessley, S. Li, H. Childs, HashFight: A Platform-Portable Hash Table for Multi-Core and Many-Core Architectures, in: IS&T International Symposium on Electronic Imaging, 2020, pp. 376–1–376–12.
[53] I. Wald, et al., Embree: A Kernel Framework for Efficient CPU Ray Tracing, ACM Transactions on Graphics (TOG) 33 (4) (2014) 1–8.
[54] S. P. Callahan, M. Ikits, J. L. D. Comba, C. T. Silva, Hardware-Assisted Visibility Sorting for Unstructured Volume Rendering, IEEE Transactions on Visualization and Computer Graphics 11 (3) (2005) 285–295.
[55] S. G. Parker, et al., OptiX: A General Purpose Ray Tracing Engine, ACM Transactions on Graphics (TOG) 29 (4) (2010) 1–13.
[56] J. Clyne, P. Mininni, A. Norton, M. Rast, Interactive desktop analysis of high resolution simulations: application to turbulent plume dynamics and current sheet formation, New Journal of Physics 9 (8) (2007) 301.
[57] H. A. Carr, G. H. Weber, C. M. Sewell, J. P. Ahrens, Parallel peak pruning for scalable smp contour tree computation, in: IEEE Symposium on Large Data Analysis and Visualization, 2016, pp. 75–84.

Appendix A. Analysis of VTK-m's Source Code

Sections 4.1 and 4.2 use a corpus documenting what MCD^3 are used in VTK-m source code. This corpus was generated using VTK-m's logging mechanism. VTK-m's logging records each time an algorithm or an MCD^3 is executed. The log further records the execution of each algorithm or when MCD^3 starts and ends so that these calls can be nested to identify when a unit is called from within another unit.

We ran VTK-m's regression test for each algorithm while recording the performance logs. We then used a Python script to read the log, identify when an algorithm was run, and generate a record of when an algorithm used MCD^3. Nested MCD^3 calls were ignored. For example, we did not record when a meta-DPP called a DPP as the intention of this analysis is to determine what MCD^3s an algorithm uses, not how meta-DPPs are implemented. If an algorithm internally used another algorithm, then any MCD^3 calls were attributed to the inner algorithm and not for the outer algorithm.

After processing the logs, the records of algorithm MCD^3 use were aggregated to generate a table that identified for each algorithm which MCD^3s it used. For each algorithm/MCD^3 combination a binary value of "yes, it was used" or "no, it was not used" was created. It should be noted that an algorithm may use a different combination of MCD^3 depending on the type of data it processes or the parameters chosen. We did not attempt to differentiate the different call parameters of algorithms. Rather, we recorded whether an MCD^3 was employed by any algorithm configuration.

Supplemental Material

This supplemental material contains more detail on aspects of the MCD^3 approach, specificially descriptions of the DPPs we use, implementation details for the meta-DPPs and convenience routines, and an example of algorithm development using MCD^3.

DPP Descriptions

This section contains descriptions of the DPPs in our MCD^3 system.

Copy Copies the data from one array to another array. This can include copying data between arrays with different structures. Although this is a DPP because it involves operations in a device memory space, it is a decidedly simple one. It is also an operation that would undoubtedly exist in any data-centric system even if it was not based on DPP operations. Thus, in our analysis of Section 4.1, we mark the use of Copy as a special operation that qualifies as DPP, but only just so.

CopyIf Given an array of values and a stencil array, copies all values that match the criterion of the stencil.

CopySubRange Copies a contiguous region of data from one array to a specified location in another array.

CountSetBits A special form of reduction that returns the total number of 1 bits in a bit field.

Fill Sets all items in a bit field to a specified value.

Reduce Apply a binary operation to all elements in an array and return the result. A typical use of Reduce is to sum all the values in an array, but other operations like finding the minimum or maximum value are possible. VTK-m also provides a variation of Reduce called **ReduceByKey**. This version additionally takes an array of keys and partitions the data by regions where adjacent keys are the same. The Reduce is then applied separately to each partition and placed in an array containing an entry for each partition.

Scan Provides a "running sum" of values in the array. That is, each item in the output array is the sum of all previous values in the input array. Addition is the most common scan operation, but others like multiplication are possible, too. Scan is used extensively in VTK-m for building indices. There are several variations of the Scan DPP. **ScanExclusive** sets the first value of the output to the initial value of the operation (usually 0), and every subsequent value is the sum of all values before the index *excluding* the current value. **ScanInclusive** sets every value to the sum of all values before the index *including* the current value. **ScanExtended** is a combination of ScanExclusive and ScanInclusive where the output array is made one larger than the input. The first entry of the output is set to the initial value and the last entry is set to the total sum. VTK-m also includes **ScanExclusiveByKey** and **ScanInclusiveByKey** that take an additional key array. Like ReduceByKey, these methods partition the input by regions where consecutive keys are equal and performs the scan independently on each partition.

Search Finds values in a sorted array. This DPP takes a group of keys to find in the sorted array simultaneously. There are two flavors of Search: **LowerBounds** and **UpperBounds**. If the sorted array contains multiple copies of the key, these methods will respectively find the first such element and the last such element.

Sort Rearranges the values in an array to be in ascending order. The variation **SortByKey** also takes a key array used to partition the input array. Each partition of the array is sorted separately.

Transform Applies a binary operation element-wise to two input arrays. The binary operation is provided as a functor that is executed for each pair of input elements.

Unique Removes all duplicate values in an array leaving only unique values. This operation only works if all duplicated values are adjacent in the input array, which can be achieved by sorting the array first.

Meta-DPP Implementations

Visit Point With Cells: Before calling the functor, this meta-DPP references the parameters to the point being visited. Retrieving point properties changes depending on the structure of the cell set (e.g. an implicit regular grid versus explicit cell indices). This is internally managed via templated programming.

The point information contains indices that can be used to retrieve fields of the incident cells. Points in a mesh have different numbers of incident cells, which complicates passing the associated data to the functor. Using a dynamically allocated array or a large allocation on the stack is not feasible on many GPU devices. Instead, the functor is passed a C++ object that behaves like an array but actually retrieves data for each incident cell on demand.

Another complication is that it is common to explicitly define a cell set by listing the points incident on each cell, but the reverse connection from points to cells is not captured. If the links from points to cells are not provided, VTK-m will automatically generate this information from the links from cells to points. These links are built using the base DPPs internally within VTK-m before the meta-DPP schedules itself. The procedure ComputePointLinks is listed in Figure A.13. Overall, ComputePointLinks uses 5 DPPs: 2 **Copy**s to initialize an array (lines 3 and 8), a **Schedule** to count cells incident on each point (line 4), a **ScanExtended** to compute the offsets into the link array (line 6), and a final **Schedule** to fill the link array (line 9). It should be noted that this is not the only way to implement ComputePointLinks, but we found that this method using atomic additions works best across the devices we tried.

After ComputePointLinks is called, the **Visit Point With Cells** meta-DPP still needs to call **Schedule** one more time to invoke the user-provided function on the cells. This brings the total DPP calls to 6.

```
ComputePointLinks(
        cellPointCount, cellLinks, cellLinkOffsets, numPoints)
 1  numCells = cellPointCount.size
 2  numLinks = cellLinks.size
    // Simple way to allocate/initialize array.
 3  Copy(ConstantArray(0, numPoints), pointCellCount)
    // Parallel for is implemented with the Schedule DPP.
 4  parallel for i = 0 to numCells
        // Atomic add counts cells referencing each point.
 5      AtomicAdd(pointCellCount, cellLinks[i], 1)
    // Convert counts to offsets.
 6  pointLinkOffsets = ScanExtended(pointCellCount)
 7  Allocate(pointLinks, numLinks)
    // pointCellCount is correct, but we need to redo the
    // atomics to find unique indices in pointLinks.
 8  Copy(ConstantArray(0, numPoints), pointCellCount)
 9  parallel for i = 0 to numCells
10      pointId = cellLinks[i]
        // Binary search index in offsets to find cell id.
11      cellId = UpperBound(cellLinkOffsets, i) - 1
        // Atomic add returns value before add.
12      localOffset = AtomicAdd(pointCellCount, pointId, 1)
13      globalOffset = pointLinkOffsets[pointId]
14      pointLinks[globalOffset + localOffset] = cellId
15  return (pointCellCount, pointLinks, pointLinkOffsets)
```

Figure A.13: Procedure to compute which cells are incident on each point based on the points incident on each cell. Explicit cell connections are represented by 3 arrays: an array identifying how many points are incident on each cell (*cellPointCount*), an array packing together the variable length sets of point ids incident on each cell (*cellLinks*), and an array containing the offset of each set in *cellLinks* (*cellLinkOffsets*). The inverse linking of cells incident on each point is represented with a similar set of arrays.

Visit Cell With Points: Before calling the functor, this meta-DPP references the parameters to the cell being visited. Retrieving cell properties changes depending on the structure of the cell set (e.g. an implicit regular grid versus explicit cell indices). This is internally managed via templated programming.

The cell information contains indices that can be used to retrieve coordinates and other fields of the incident points. Cells of different types have different numbers of points, which complicates passing the associated data to the functor. Using a dynamically allocated array or a large allocation on the stack is not feasible on many GPU devices. Instead, the functor is passed a C++ object that behaves like an array but actually retrieves data for each incident point on demand.

Point Neighborhood: This meta-DPP only works with structured cells. Before calling the functor provided to it, the meta-DPP computes the envelope of accessed points based on the requested network. Special objects are passed to the provided functor that allow a point field value to be retrieved based on indices relative to the visited point.

Reduce By Key: Before this meta-DPP schedules itself, it identifies duplicate keys in its key array input and builds index arrays to point to associated values. The procedure BuildKeyArrays to create these index arrays is listed in Figure A.14. After initializing an index array with a **Copy** (line 2), BuildKeyArrays sorts the keys with a **SortByKey** to trace back to the original indices (line 3). It then uses the basic **ReduceByKey** to count the number of times each key is used and to get an array of unique keys (line 4). Finally, **ScanExtended** is used to derive key group offsets in the index array (line 5).

```
BuildKeyArrays(keys)
1  numKeys = keys.size
   // Initializes valueMap to 0, 1, 2, ...
2  Copy(IndexArray(numKeys), valueMap)
   // Group like keys and build a map to original index
3  SortByKey(keys, valueMap)
   // Count the number of times each key occurs
4  (uniqueKeys, counts) = ReduceByKey(
                keys, ConstantArray(1, numKeys), Sum)
   // Convert counts to offsets.
5  offsets = ScanExtended(counts)
6  return (counts, valueMap, offsets)
```

Figure A.14: Procedure to identify groups of common keys and produce index arrays to the indices containing each key.

Given these pre-computed arrays, the **Reduce By Key** meta-DPP follows the built links from reduced value to duplicate key group to the values in the source arrays to extract information to pass to the functor. The number of values associated with each unique key will vary, so like the other meta-DPPs, the Reduce by Key meta-DPP passes a C++ object that behaves like an array but actually retrieves data for each element in the group.

Overall, a call to the **Reduce By Key** meta-DPP encapsulates 6 DPPs. In practice, the call first performs a **Copy** to preserve the key array provided. It then expends 4 DPPs running the BuildKeyArrays procedure. And finally, it **Schedule**s the functor provided by the user.

Map Field: This meta-DPP is similar in behavior to the base Transform DPP, which applies a functor as a binary operation to two input arrays to produce a new array. However, the base Transform DPP has a rigid structure of two input arrays and a single output array, which makes operations leveraging different numbers of arrays difficult to implement.

Additionally, the Map Field meta-DPP, like all other meta-DPPs, work with the modifies described in Section 3.2.6, which allow the meta-DPP to work with the jagged inputs and outputs typical of visualization algorithms.

The objective metrics collected in Section 4.1 suggest that the added benefits of the Map Field meta-DPP over the base DPPs are useful to developers.

Scatter: The behavior of all the previously described meta-DPPs can be modified by scatter and map structure. The most common non-trivial modifier is the **Scatter Counting**, which takes an array of the number of outputs to be generated by each input. As described in Section 3.2.6, a **Scatter Counting** can be used to either add or remove data from the input.

Any scatter modifier needs to provide an *outputToInputMap*

that identifies for each output which input it comes from and a *visitArray* that identifies for each output an identifier that is different than the visit index for any other output sharing the same input. The **Scatter Counting** modifier derives these arrays from a provided array of counts for each input value as described by the BuildScatterArrays in Figure A.15. Building these arrays requires 4 DPPs: a **ScanInclusive** to map input to output (line 1), a **UpperBounds** to invert this map (line 2), a **LowerBounds** to identify where each output group starts (line 3), and a **Schedule** to fill the visit array (line 5).

```
BuildScatterArrays(counts)
1  (inputToOutputMap, outputSize) = ScanInclusive(counts)
   // Determine the input associated with each output.
2  outputToInputMap = UpperBounds(
                inputToOutputMap, IndexArray(outputSize))
   // Determine the index to the start of each group.
   // Repeated items will all point to the first item.
3  startOfGroups = LowerBounds(
                outputToInputMap, outputToInputMap)
4  Allocate(visitArray, outputSize)
   // Parallel for is implemented with the Schedule DPP.
5  parallel for i = 0 to outputSize
6      visitArray[i] = outputToInputMap − startOfGroups
7  return (outputToInputMap, visitArray)
```

Figure A.15: Procedure to map output indices to input indices given an array of the number of outputs each input produces.

Convenience Routine Implementations

ArrayRange: Computing the range of an array is a simple application of using the **Reduce** DPP to compute the minimum and maximum values of an array. A trivial implementation would call **Reduce** twice, once to compute the minimum and once to compute the maximum. That said, the base VTK-m implementation plays tricks with the operator and data type to compute both minimum and maximum with one call to **Reduce**. Thus, we count the **ArrayRange** as a DPP complexity of 1 even though a simple implementation would use more.

CountToOffset: As described in Section Appendix A, structures like explicit cell sets and key groups are represented by a packed array of sets with an array of offsets to each set. However, it is almost always the case when constructing such a structure that you have an array of set sizes (from counting the number of entries in each), but no offsets. So converting this array of counts to an array of offsets is a common operation. This is easily implemented with a **ScanExtended** operation on the counts, and VTK-m provides a convenient wrapper for this.

Locators: A locator takes a point coordinate in 3-space and determines which cell from a data set contains that coordinate. VTK-m provides several locator types whose use depends on the type of cell structure and operation. The most general locator in VTK-m is a 2-level grid algorithm based on the work of Kalojanov, Billeter, and Slusallek [40]. We refer the reader to that work for details on implementation, but we note that the implementation in VTK-m requires the use of 18 DPPs.

MapFieldMergeAverage: When a VTK-m filter modifies the topology of a data set, it needs to modify the field arrays of the input to match the topology of the output. For example, a filter might modify a topology by merging together points that are nearby as part of a cleaning or coarsening operation. The field values of the merged points need to be combined in some way, and averaging the values is a common method to do that. Thus, VTK-m provides a convenience function to merge field values using the average operator. This routine is implemented with a straightforward application of the **Reduce By Key** meta-DPP, which itself uses 6 DPPs.

MapFieldPermutation: Another common change in topology is to reorder points or cells in the data, in which case the field values need to be similarly reordered. For example, the threshold filter will remove some cells based on a selection criteria, and the remaining cells will be packed. Thus, the cell field data needs to be similarly repacked. Thus, VTK-m provides a convenience function to reorder field arrays based on the permutation created by the filter. The implementation is a simple **Schedule** that uses the permutation to retrieve the correct input value for each output.

Example MCD^3 Algorithm Development

To demonstrate the value of our MCD^3 approach, we provide a simple example of using them for a basic scientific visualization algorithm. Note that to make the description easier to follow, we use pseudocode for functors and a description of how they interact with meta-DPPs. Details on operating the meta-DPPs in VTK-m can be found in the software documentation [42].

As a simple example of how MCD^3 simplifies algorithm development, let us consider estimating the normals of a mesh surface. Normals, which are unit vectors pointing perpendicular to a surface, are extremely important for lighting calculations in 3D graphics. Surfaces in scientific visualization are usually represented as a mesh of flat polygons. Properly estimating normals allows the rendering to draw the polygons more like the smooth surface they are supposed to represent.

A simple estimation of smooth normals for a polygonal mesh, regardless of whether parallel or serial, requires two steps. The first step is to compute the normal to each flat polygon, and the second step is to average these normals on the shared vertices of the mesh.

This first step is easily achieved with the Visit Cell with Points meta-DPP. Figure A.16 shows a functor, PolygonNormalFunctor, that takes the vertices of a polygon and uses the cross product to find a vector perpendicular to it. When this functor is used with the Visit Cell with Points meta-DPP, it will generate a cell field array containing vectors perpendicular to the surface.

For a rendering system to depict a smooth surface, it really needs normals attached to the points of the mesh so they can be interpolated across the polygons. The Visit Cell with Points

PolygonNormalFunctor(*vertexCoordinates*)

```
1  v1 = vertexCoordinates[2] − vertexCoordinates[1]
2  v2 = vertexCoordinates[0] − vertexCoordinates[1]
3  n = Cross(v1, v2)
4  return n/||n||
```

Figure A.16: Functor to compute to normal to a single polygon. When used with the Visit Cell with Points meta-DPP, each functor instance receives the coordinates for all the vertices of the polygon.

meta-DPP with PolygonNormalFunctor has just produced normals on each polygon. To move the normals from the polygons to the points, we can use the Visit Point with Cells meta-DPP with AverageNormalsFunctor shown in Figure A.17.

AverageNormalsFunctor(*cellNormals*)

```
1  aggregateN = [0, 0, 0]
2  for each n ∈ cellNormals
3      aggregateN = aggregateN + n
4  return aggregateN/||aggregateN||
```

Figure A.17: Functor to average normals vectors. When used with the Visit Point with Cells meta-DPP, each functor instance receives the normals for all cells incident on a point.

When the Visit Point with Cells meta-DPP is given AverageNormalsFunctor, the polygon mesh, and the previously computed cell normals, it will provide a point field array containing surface normals.

Although performing these operations is very simple with MCD^3, it is quite complicated with only a basic DPP system. The base DPPs have no direct way to collect the vertex information on each cell. The easiest way to achieve this with a DPP is to use a ParallelFor to create a kernel to dereference random access array with indices. This requires much more code and is more error prone as it does not benefit from the thread safety provided by DPP and meta-DPP routines.

The second part of the algorithm, averaging the cell normals to each point, is even more difficult with basic DPPs. A typical polygon mesh representation captures the points incident on each cell but does not directly express the cells incident on each point. Deriving this information requires enumerating all points of all cells, reordering this enumeration to collect like points, and rebuilding the variable index lists.

Furthermore, an implementation using basic DPPs would be specific to a particular layout of the data. The assumption of the previous description is that a polygonal mesh is represented by explicitly declaring which point each cell uses. This is typical, but other representations are possible. For example, one might represent a surface using a 2D regular grid of quadrilateral cells with a height field. Our MCD^3's data model integrated with the meta-DPPs allows our single implementation to work with both of these representations and more.

License