LA-UR-13-23809

Approved for public release;
distribution is unlimited.

Title: Portable Data-Parallel Visualization and Analysis in Distributed Memory Environments

Author(s): Christopher Sewell
Li-ta Lo
James Ahrens

Intended for: IEEE Symposium on Large-Scale Data Analysis and Visualization, 2013-10-13/2013-10-14 (Atlanta, Georgia, United States)

Portable Data-Parallel Visualization and Analysis in Distributed Memory Environments

Christopher Sewell* Li-ta Lo† James Ahrens‡

CCS-7, Los Alamos National Laboratory

ABSTRACT

Data-parallelism is a programming model that maps well to architectures with a high degree of concurrency. Algorithms written using data-parallel primitives can be easily ported to any architecture for which an implementation of these primitives exists, making efficient use of the available parallelism on each. We have previously published results demonstrating our ability to compile the same data-parallel code for several visualization algorithms onto different on-node parallel architectures (GPUs and multi-core CPUs) using our extension of NVIDIA's Thrust library. In this paper, we discuss our extension of Thrust to support concurrency in distributed memory environments across multiple nodes. This enables the application developer to write data-parallel algorithms while viewing the data as single, long vectors, essentially without needing to explicitly take into consideration whether the values are actually distributed across nodes. Our distributed wrapper for Thrust handles the communication in the backend using MPI, while still using the standard Thrust library to take advantage of available on-node parallelism. We describe the details of our distributed implementations of several key data-parallel primitives, including scan, scatter/gather, sort, reduce, and upper/lower bound. We also present two higher-level distributed algorithms developed using these primitives: isosurface and KD-tree construction. Finally, we provide timing results demonstrating the ability of these algorithms to take advantage of available parallelism on nodes and across multiple nodes, and discuss scaling limitations for communication-intensive algorithms such as KD-tree construction.

Index Terms: D.1.3 [Programming Techniques]: Concurrent Programming—Parallel programming

1 INTRODUCTION

Gains in the number of operations per second achievable with new hardware architectures have been increasingly driven by higher degrees of concurrency rather than by increasing clock speeds. Nevertheless, the heterogeneity of available architectures generally requires a significant amount of platform-specific optimization of code, taking into account such specifics of the target architecture as the structure of the cache hierarchy, the number and organization of available threads and thread groups, the vector width, etc., in order to take advantage of the available parallelism and approach the theoretical potential. When an architecture supports a cross-platform language, such as OpenCL, it may be possible to simply run existing code on a new architecture, but the code must still be tuned and often re-designed in order run efficiently.

Data-parallelism is a programming model that maps well to architectures with a high degree of concurrency. Using only a limited set of "embarrassingly parallel" data-parallel primitive operators, such as scan, transform, and reduce, a wide variety of higher-level algorithms can be constructed. These algorithms will then run efficiently on any architecture for which an efficient implementation of the data-parallel primitives exists, without needing to optimize the higher-level algorithm for the target platform.

In [1], we introduced PISTON, our framework for developing data-parallel visualization and analysis operators. PISTON is built on top of NVIDIA's Thrust library. Thrust is a C++ template library that provides CUDA, OpenMP, and TBB implementations for a set of simple algorithms, most of which are data-parallel primitives [2]. These algorithms operate on host and device vectors (where the device may be an accelerator such as a GPU or an Intel Xeon Phi), and data can be transferred between the two data types. We implemented an isosurface algorithm using these data-parallel primitives, based on the standard Marching Cubes algorithm [3], and designed to minimize data movement by creating a reverse mapping from output vertex index to input cell index. Cut surface and threshold operators using the same algorithmic pattern were also presented. We showed that our data-parallel implementations of these algorithms allowed us to compile the exact same operator code for different multi and many-core architectures (GPUs using Thrust's CUDA backend and our prototype OpenCL backend, and multi-core CPUs using our enhancement of Thrust's OpenMP backend). The performance cost for this portablilty was shown to be fairly small relative to reference platform-native implementations for these algorithms.

While our previous work focused on taking advantage of on-node parallelism in shared memory environments, in this paper we extend our data-parallel framework to also work across nodes in a distributed memory environment. Many key data-parallel primitives have been implemented in a new wrapper backend, which uses MPI to communicate between nodes. Within the wrapper backend, the existing CUDA, OpenMP, or TBB backend implementations are called for on-node work in order to also still take advantage of all cores available on each node. Higher-level algorithms can then be written in the same manner as the data-parallel algorithms we presented in our previous work for a single node. The data can be treated as a single vector (even though it is actually distributed across nodes), with the inter-node communication handled "under the hood" in the distributed backend. Thus, virtually identical operator code can be compiled and run efficiently on a single multi-core CPU or many-core accelerator (such as a GPU), or a set of nodes with multi-core CPUs or many-core accelerators.

In this paper, we first describe relevant related work. Then, we describe the implementation details of a number of key data-parallel primitives in our distributed backend. Higher-level algorithms for isosurface and KD-tree construction are then presented, each built using these distributed data-parallel primitives. In the Results section, we compare the performance of the isosurface implemented using the distributed data-parallel primitives to three alternatives: manually dividing the input equally among the nodes, each of which runs our on-node isosurface algorithm; using parallel VTK; and performing the computation for the entire input on a single node. We also demonstrate scaling and performance results for the KD-tree construction algorithm. Finally, the implications of

*e-mail:csewell@lanl.gov

†e-mail:ollie@lanl.gov

‡e-mail:ahrens@lanl.gov

this work and some opportunities for future work are summarized in the Conclusion.

2 Related Work

The theoretical foundation for our work in data parallelism is laid out in [4]. Blelloch describes a scan vector model, consisting of a set of data-parallel primitives very similar to those now available in Thrust. He then outlines a variety of higher-level algorithms constructed using this scan vector model in the fields of data structures, computational geometry, graphs, and numerical analysis. Our KD-tree construction algorithm is based on the outline provided by Blelloch. Nested data parallelism, as found in, for example, divide-and-conquer algorithms such as quicksort and KD-tree construction, is flattened using segment descriptors. Blelloch's data parallel programming model efforts were implemented on the Thinking Machine's Connection Machine family of hardware (CM2, CM-200 and CM5) [5]. Johnsson led a team of computer scientists and applied mathematicians to optimize algorithms and applications on this data parallel hardware architecture [6, 7]. Blelloch's NESL functional language, designed to support nested data parallelism on wide vector machines, was ported to GPUs by [8]. Our work significantly extends these efforts by providing a portable implementation of data parallel constructs on modern multi-core and accelerator-based architectures.

More generally, a number of people have made significant progress in implementing algorithms, including many used in visualization and analysis, on high-concurrency distributed memory and mixed shared-distributed memory architectures. For example, [9] describes the implementation of a parallel volume renderer within the VisIt library that has been run with large data sets (8192^3 voxels) on up to 256 GPUs. Similarly, [10, 11] extend the CUDA parallel programming model to the PCI bus and network interconnects, facilitating the use of multiple GPUs. They provide a set of language extensions called CUDASA, which presents distributed shared memory to the programmer as global arrays [12]. Others have explored the advantages and disadvantages of mixed-mode programming using both MPI and OpenMP, identifying situations in which such a strategy is most efficient (such as when a pure MPI program suffers from poor scaling due to load imbalance or too fine a grain problem size) [13]. Libraries, such as Argonne National Laboratory's DIY [14], have been developed in order to provide cleanly encapsulated implementations of common MPI functionality and communication patterns, allowing algorithm developers to write code without having to directly deal with the complexities of distributed memory programming. In addition to Thrust, other libraries, such as STAPL, have been developed to provide an STL-like interface for parallel programming [15]. Data parallelism has also been further explored in the context of extending the VTK visualization library to take advantage of many and multi-core architectures in the Dax [16] and EAVL [17] projects. The OpenMPI [18] and related OpenRTE [19] projects have attempted to improve the MPI runtime to work well on large, heterogeneous systems, with ease of use, resiliency, scalability, and extensibility as design objectives. MapReduce [20] is a popular programming model for processing large data sets across distributed, heterogenous clusters, although the class of problems that can be solved in this model is relatively limited.

Our research makes several significant contributions to extend this existing body of work. Embedding both the distributed and shared memory data parallelism within a framework based on Thrust allows the algorithm developer to program using C++ and an STL-like vector library, which is likely to be more familiar, easier to use, and simpler to integrate with other code than most alternatives. In contrast to much of the related work which is focused on a single architecture and/or language (e.g., CUDA on NVIDIA GPUs), our framework is portable across different types of on-node parallelism (GPUs, CPUs, MICs, etc.). By supporting nested parallelism through the use of segmented vectors, it allows for a wide variety of algorithms to be efficiently implemented. Finally, it allows the programmer to write an algorithm in the same way regardless of whether it is to be run serially, with any supported type of shared-memory parallelism, with distributed-memory parallelism, or with both distributed and shared-memory parallelism.

3 Implementation

Our distributed backend is currently implemented as a wrapper around the standard single-node Thrust. Like Thrust, the code is provided in header files (.h and .inl), is contained within a namespace called `dthrust`, and provides an interface of functions with similar or identical names and signatures as corresponding Thrust functions. This makes it easy to still compile and use Thrust for various on-node accelerators while using the distributed wrapper. For example, a scan across multiple GPUs could be called using `dthrust::inclusive_scan` instead of `thrust::inclusive_scan`, and compiling with the CUDA backend. The operator code is written as if there were a single device vector, while in fact the elements of the vector are spread across device vectors in each process. In this section, we describe our implementations of several of the more important data-parallel primitives, as well as several higher-level algorithms built using these primitives.

3.1 Distributed Implementations of Data-Parallel Primitives

3.1.1 Data Transfer Functions

In some cases, all of the data may fit in the host memory of the root process, and the reason for distributing the data among multiple ranks is to take advantage of multiple accelerators (such as GPUs) which have their own memory spaces (which are likely smaller than that of the host). For such cases, utility functions are provided to easily transfer data from a host vector on the root to device vectors in each process and vice-versa. This can also be helpful for debugging code that will later be run with a larger input data set that is itself distributed across processors. The `device_to_host` function uses an `MPI_Gather` to collect the vector sizes from each process, performs an exclusive scan on the vector sizes to get the displacements into the root host vector at which to begin writing data from each process, transfers the local data from device to host, and finally uses an `MPI_Gatherv` to send data from all processes to the root host vector. The `host_to_device` function sets the vector size for each process so as to divide the host vector evenly (with any remainder on the last process), or to user-specified sizes on each processor, then uses an `MPI_Gather` to collect the vector sizes on each process, performs an exclusive scan to get the displacements into the root host vector at which to begin reading the data for each process, uses an `MPI_Scatterv` to send data from the root host vector to each process, and finally transfers the local data on each process to a device vector.

3.1.2 Rebalancing and Shifting

A rebalance function allows a vector to be re-balanced across the processors, either uniformly, or so as to match the distribution of another vector of equal global length. The first case provides better load balancing, while the second case is useful in order to perform another function (such as transform) on two or more vectors while keeping all computation local. The current local vector sizes are gathered to the root, which performs an exclusive scan to compute the current global offset for each processor, which it scatters to the respective processors. The new desired local sizes for each processor (either computed to be uniform, or taken from the reference vector on that processor) are broadcast to all processors, so that all processors can perform an exclusive scan and get the new global

starting indices for each processor. A counting iterator starting with the processor's current global offset can then be searched in the vector of new global offsets using the regular Thrust `upper_bound` function to get the new processor id to which each element will belong. Each processor then informs each other processor how many elements it will send to it using an `MPI_Scatter`. Finally, in order, each processor uses an `MPI_Scatterv` to send data belonging to each other processor, and each receiving processor appends the received data to its new local vector, while the sending processor appends its local data to its new local vector, ensuring that the elements in the result are in the correct order (data from lower-numbered processors should come before data from higher-numbered processors). In most cases, each processor will only need to communicate with a small number of other processors with ranks close to its own, but in the worst case, any processor may need to send data to any other processor.

The implementation of the shift function also begins by gathering the local vector sizes to the root, which performs an exclusive scan to compute the total global size and the global offset for each processor, which it scatters to the respective processors. (Alternatively, this may be accomplished with an `MPI_Exscan`.) Given its global offset, the global size, and the number of places to shift, each processor can compute how many, if any, of its elements should be dropped rather than copied to the local output vector. The first processor (in the case of a right shift), or the last processor (in the case of a left shift), append a number of zeros to its local output vector equal to the number of shift places. As with some of the other operators to be described later, this algorithm limits communication (no elements are actually moved), at the expense of ensuring good load balance. However, if desired, the result can then be fed to the rebalance function described above, which will optimize the load balance at the expense of some communication.

Functions for other utility functions such as creating counting iterators and for returning the first (front) or last (back) element of the global vector are also included.

3.1.3 Transform, Scan, Segmented Scan, Scatter, Gather, and Reductions

Unary transforms can be performed locally on each processor so long as user-defined functors either do not access any other vectors, or are restricted to only access vectors with an equivalent distribution across processors (which can be ensured with the `rebalance` function). Similarly, a binary transform can be performed trivially on two vectors with an equivalent distribution.

Inclusive and exclusive scans for a given binary operator may be performed by executing the scan on each local device, gathering the local sums (where "local sum" means the final result of the local scan, using the given binary operator, combined with the final local input value in the case of an exclusive scan), performing a scan on those, and then applying each value in this result as an offset for the corresponding process. The scan on the processor sums can be performed using the `MPI_Exscan` function, but we have found that performing this scan by using the `MPI_Gather` function, executing the scan on the root, and using `MPI_Scatter` to distribute the computed offsets back to the processors is slightly faster and easier to adapt to special cases, such as reverse iterator input (in which case the scan needs to be performed in reverse). The distributed scan algorithm is illustrated in Figure 1, and code is given in Listing 1.

For segmented scans (called `scan_by_key` in Thrust), the segmented scan is also first executed locally on each device, and the local sums, as well as the first and last segment id for each local vector, are gathered on the root. However, since each processor may contain more than one segment, and each segment may span more than one processor, the processor offsets cannot be computed by simply performing one scan on the processor sums. Instead, for each processor, the previous processor sums are accumulated so long as their last segment id is equal to its first segment id. In the worst case, this could take $O(p^2)$ time, where p is the number of processors. These offsets are then distributed to each processor using an `MPI_Scatter`, and each processor applies the received offset only to its elements with the same segment id as the last element of the previous processor, using Thrust's `transform_if` function.

A scatter operation is implemented by first broadcasting the number of elements belonging to each processor and computing an exclusive scan on this result to obtain the global index at which each processor's vector begins. A functor which takes this vector of processor start indices as input is used with a transform operator to compute the processor destination and local index from each global destination index. Each processor can then either perform a `scan_by_key` to order the input data and local indices by destination processor and then use `MPI_Scatterv` to send all this data to each other processor, or can avoid the sort and use `copy_if` to stream compact the values to be sent to each other processor one at a time using `MPI_Send` and `MPI_Recv`. Finally, Thrust's local `scatter` operator can be used to locally distribute the data to their proper local indices. While this distributed scatter operator can often be a useful tool in designing higher-level data-parallel algorithms, it can incur significant communication overhead (in the worst case, all elements are sent once to another processor), so it should only be used when needed.

A gather operation (which is also performed by a permutation iterator in Thrust) can be implemented in a similar manner. However, once the global map indices have been converted to processor ids and local indices, two phases of communication are necessary: the first for the requesting processors to send local indices to be fetched from other processes, and the second for these processes to return the requested data. Local data residing on the same processor as the one where it was requested are copied to the output using a `gather_if` (conditioned on the computed source process rank being equal to the process's own rank). Data requested from the other processes is received back in the order of the source processors' ranks, so the final local destination indexes for the vector of received data values must be computed by stream compacting a counting iterator using a `copy_if` conditioned on the source process not being equal to the process's own rank and then performing a `stable_sort_by_key` (with the source process ids as the keys). The vector of received data can then be scattered to these local indices in the output.

Distributed reductions are implemented simply by performing the reduction locally on each processor, gathering the results of these local reductions to the root, and performing a final global reduction on these values. To match the syntax of Thrust, transform scans are also provided, although those are currently implemented simply as a transform followed by a scan without kernel fusion.

3.1.4 Upper and Lower Bound

A restricted version of the Thrust search operator `upper_bound` has also been implemented in the distributed wrapper. As with the regular Thrust `upper_bound`, the input search vector must be an ordered sequence, but in this case, the vector of values for which to search must be a counting iterator, and each element of the ordered sequence must be either equal to or one greater than the previous element. This restricted operator is useful for many applications, as shown, for example, in the distributed isosurface algorithm described below. Each processor can use the regular `upper_bound` operator to determine the local indices for all counting iterator values between the minimum and maximum values of the portion of the ordered sequence vector on that processor. Each processor (except the last) also communicates the last value in its portion of the ordered sequence to the next processor. If this value is not the same

```
template <typename InputIterator, typename OutputIterator, typename T, typename BinaryOperation>
OutputIterator scan(InputIterator first, InputIterator last, OutputIterator result, T init, bool inclusiveScan, BinaryOperation binop, bool reverse)
{
    // Get the MPI rank, total number of processes, MPI data type, and local vector size
    int commSize; MPI_CHECK(MPI_Comm_size(MPI_COMM_WORLD, &commSize));  int commRank; MPI_CHECK(MPI_Comm_rank(MPI_COMM_WORLD, &commRank));
    MPI_Datatype dataType;  int dataTypeFactor;  dthrust::get_mpi_type<T>(typeid(T), dataType, dataTypeFactor);  int N = last - first;

    // Perform the inclusive or exclusive scan locally on this processor
    if (inclusiveScan) thrust::inclusive_scan(first, last, result, binop);
    else if (commRank == 0) thrust::exclusive_scan(first, last, result, init, binop);
    else thrust::exclusive_scan(first, last, result, 0, binop);

    // Get the local sum (the last element of the scan, combined with the last element of the input in the case of an exclusive scan)
    T localSum = init;  if (N > 0) localSum = *(result+N-1);
    if ((!inclusiveScan) && (N > 0)) localSum = binop(localSum, *(last-1));

    // Gather local sums to the root, perform a scan on those, and scatter these offsets
    thrust::host_vector<T> localSums;
    if (commRank == 0) { localSums.resize(commSize+1); thrust::fill(localSums.begin(), localSums.end(), init); }
    MPI_CHECK(MPI_Gather(&localSum, 1, dataType, thrust::raw_pointer_cast(&*(localSums.begin() + (reverse ? 0 : 1))), 1, dataType, 0, MPI_COMM_WORLD));
    if (commRank == 0)
      if (reverse) thrust::inclusive_scan(localSums.rbegin(), localSums.rend(), localSums.rbegin(), binop);
      else thrust::inclusive_scan(localSums.begin(), localSums.end(), localSums.begin(), binop);
    MPI_CHECK(MPI_Scatter(thrust::raw_pointer_cast(&*(localSums.begin() + (reverse ? 1 : 0))), 1, dataType, &localSum, 1, dataType, 0, MPI_COMM_WORLD));

    // Apply the received offset to each element on this processor
    thrust::transform(result, result+N, thrust::make_constant_iterator(localSum), result, binop);
    return (result+N);
}
```

Listing 1: Code for distributed scan operator

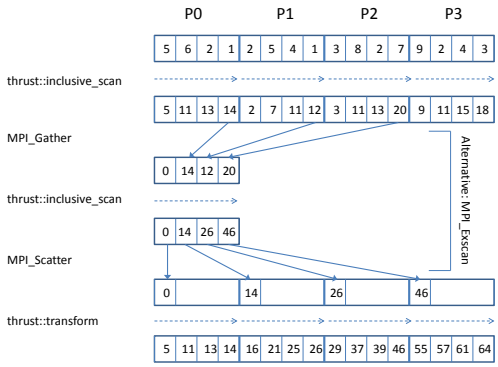

Figure 1: Diagram of the distributed scan algorithm (showing here an inclusive scan)

as the first value owned by the receiving process, the receiving process adds an extra value at the beginning of its portion of the output vector with a local index of zero. Using an `MPI_Exscan` on the local ordered sequence vector size, it can compute the beginning global index of the ordered sequence on each processor, and use the transform operator to add this global offset to the computed local indices. If the last value in the ordered sequence on the last processor is less than the maximum value of the counting iterator, the last index of the ordered sequence is appended to the output vector for each counting iterator value in excess of the final ordered sequence value.

This algorithm results in a very limited amount of inter-node communication, with each processor sending only a single value to one other process. However, it can result in an output vector that is not evenly balanced across the processors. If desired, the output vector can be passed to the `rebalance` utility function to improve load balancing. An implementation of `lower_bound` should be symmetrical to this algorithm, but has not yet been implemented.

3.1.5 Sort and Sort by Key

Our distributed wrapper implements the `sort` operator using a distributed sample sort [21] [22]. The data is first sorted locally on each processor using the regular Thrust `sort` operator. Then, equally-spaced samples are selected from the sorted data on each processor and gathered onto the root where they are sorted. Equally spaced splitter values are then selected from the sorted samples and broadcast to each processor. Each processor can then compute the ranges in its local sorted data to send to each other processor, either by calling `count_if` for each processor id and then performing an exclusive scan, or by using `upper_bound`. All data that needs to be sent to other processors is then distributed using `MPI_Scatterv`. The data received from the other processes is sorted and then merged with the local data remaining on the processor using the regular Thrust `sort` and `merge` functions. Aside from the communication to determine the splitter values, the data movement is the minimum required to get each element to its proper processor, and the load balancing is good.

Sort by key (in which the elements in a vector of values are rearranged along with their correponding keys as the keys are sorted) is implemented using the same algorithm. Thrust's `sort_by_key` is used instead of `sort` for the local sorts, and the associated values must be sent between processors along with the keys that are moved. The received keys and values as well as the local keys and values are copied to the corresponding key and value output vectors and sorted together using `sort_by_key`.

3.2 Algorithms Built Using the Distributed Primitives

Higher-level data-parallel algorithms, such as those commonly used in visualization and analysis, can then be constructed using the distributed primitives described in the previous section. In this section, we describe two such algorithms: isosurfacing, based on the algorithm we previously published using Thrust on a single multi- or many-core node, and constructing a KD-tree. The key advantage of this strategy from the point of view of the algorithm developer is that he/she can design the algorithm in the same way as he/she would for a single node or even single core essentially without needing to explicitly consider the fact that it will run on multiple cores and on multiple nodes, and yet the resulting operator will take advantage of the available parallelism and be portable across any architectures supported by the backends.

In the case of the isosurface, it is also possible to get a speedup across multiple nodes in a fairly straightforward manner without using distributed data-parallel primitives by running the single node implementation on each node, each operating on a subset of the input with ghost cells replicated as necessary (although the speedup may not be ideal due to poor load balancing of the number of output vertices). However, while distributed algorithms often exist, many other operators, such as constructing a single global KD-tree across multiple nodes, require explicit consideration of the distributed environment and cannot be implemented as a straightforward extension of a single-node shared-memory algorithm without using distributed data-parallel primitives.

```
// Initialize a counting iterator, and allocate memory for vectors
thrust::counting_iterator<int> countingIterator;  dthrust::make_counting_iterator(0, input.NCells, countingIterator);
dthrust::resize(input.NCells, case_indices);  dthrust::resize(input.NCells, num_vertices);  dthrust::resize(input.NCells, valid_cell_enum);

// Classify all cells to get Marching Cubes case index and number of vertices to generate
thrust::transform(countingIterator, countingIterator+case_indices.size(), thrust::make_zip_iterator(thrust::make_tuple(
                  case_indices.begin(), num_vertices.begin())), classify_cell(input, isovalue, numVertsTable.begin()));

// Enumerate the valid cells, and then find the indices of the valid cells
dthrust::transform_inclusive_scan(num_vertices.begin(), num_vertices.end(), valid_cell_enum.begin(), is_valid_cell(), thrust::plus<int>());
dthrust::upper_bound_counting(valid_cell_enum.begin(), valid_cell_enum.end(), num_valid_cells-1, valid_cell_indices);

// Use valid cell indices to fetch case index and number of vertices for each valid cell
dthrust::gather(num_vertices.begin(), num_vertices.end(), valid_cell_indices.begin(), valid_cell_indices.end(), output_vertices_compact);
dthrust::gather(case_indices.begin(), case_indices.end(), valid_cell_indices.begin(), valid_cell_indices.end(), case_indices_compact);

// Do an enumeration to get output indices for first vertex generated by valid cells
output_vertices_enum.resize(output_vertices_compact.size());
dthrust::exclusive_scan(output_vertices_compact.begin(), output_vertices_compact.end(), output_vertices_enum.begin(), 0, thrust::plus<int>());

// Get global and local number of vertices, and allocate space for vertex arrays
num_total_vertices = dthrust::back(output_vertices_compact) + dthrust::back(output_vertices_enum);
int num_local_vertices = 0;
if ((output_vertices_compact.size() > 0) && (output_vertices_enum.size() > 0))
    num_local_vertices = output_vertices_compact.back() + (output_vertices_enum.back() - output_vertices_enum.front());
vertices.resize(num_local_vertices);  normals.resize(num_local_vertices);  scalars.resize(num_local_vertices);

// Do edge interpolation for each valid cell
if (num_local_vertices > 0)
    thrust::for_each(thrust::make_zip_iterator(thrust::make_tuple(valid_cell_indices.begin(), output_vertices_enum.begin(),
        case_indices_compact.begin(), output_vertices_compact.begin())), thrust::make_zip_iterator(thrust::make_tuple(valid_cell_indices.end(),
        output_vertices_enum.end(), case_indices_compact.end(), output_vertices_compact.end())), isosurface_functor(input, source, isovalue,
        output_vertices_enum[0], triTable.begin(), thrust::raw_pointer_cast(&*vertices.begin()), thrust::raw_pointer_cast(&*normals.begin()),
        thrust::raw_pointer_cast(&*scalars.begin())));
```

Listing 2: Code for distributed isosurface algorithm

3.2.1 Isosurface

Our distributed isosurface algorithm is almost identical to our previously published single-node algorithm [1]. The general principle is that it generates a "reverse mapping" from output vertex index to input cell index (rather than from input cell index to output vertex index), allowing it to "lazily" apply operations only to cells that will generate the output vertices. Code for the distributed algorithm is given in Listing 2. Implementations of the functors (such as `classify_cell` and `isosurface_functor`) are not listed, but are exactly the same as those used in the single-node algorithm, except for a small modification when used with a data set from a file (rather than procedurally generated data), as described later in this section.

Using the transform primitive, with the `classify_cell` functor that computes the Marching Cubes case number index for a cell based on its pattern of vertices above and below the isovalue, a vector of case number indices (`case_indices`) and a vector of the number of output vertices generated by each cell (`num_vertices`) are generated. A `transform_inclusive_scan` with the `is_valid_cell` functor that returns one for any value greater than zero is then performed on the number of vertices to enumerate the valid cells (`valid_cell_enum`). The last element of this vector indicates the total number of valid cells. A search (`upper_bound_counting`) is then performed on a counting iterator that enumerates the valid cells, searching in the `valid_cell_enum` vector to find the index of the first element greater than each counting iterator element. The result of this search is stored in `valid_cell_indices`. This compact vector of global indices of the valid cells is used to fetch the number of output vertices for each valid cell using `gather`, and an `exclusive_scan` on this result (`output_vertices_compact`) gives the starting offset into the global output vertex array for each valid cell (`output_vertices_enum`). In the original single-node algorithm, the gather was combined with the exclusive scan using Thrust's permutation iterator, but we have not yet implemented distributed permutation iterators. The total number of vertices in the output is the sum of the last elements of the `output_vertices_compact` and `output_vertices_enum` vectors (the starting offset of the final valid cell plus the number of vertices produced by the final valid cell). One small additional step required in the distributed version of the algorithm is to compute the number of local vertices in the same way as the number of global vertices, except using only the local versions of those two vectors, in order to correctly resize the local sections of the vertex, normal, and scalar vectors. Finally, the vertices, normals, and scalars are generated using `for_each` with the `isosurface_functor` functor. Since no communication between vector elements is needed in this step, the standard single-node `for_each` may be called by each processor (as was also the case with the `transform` in the first step).

Due to the implementation of `upper_bound_counting`, as described above, the `gather` step requires virtually no communication, but is likely to result in a `valid_cell_indices` vector that is not well balanced across nodes. The vector may be evenly redistributed using the `rebalance` function, at the cost of inter-node communication. In practice, the number of cells that actually generate geometry is usually quite small compared to the original number of input cells, so an unbalanced workload in the final steps that compute the vertices for each valid cell may not significantly impact the overall performance, which may be dominated by the initial, well load-balanced steps that examine all input cells to determine which will generate geometry.

When data from a file (such as the ocean temperature data set shown in Figure 8) is used rather than procedurally generated data (such as the "tangle" field described in the Results section), the appropriate range of data must be used to instantiate the local PISTON image data structure on each processor. For VTK image files used as input to the isosurface operator, in which the layout of the 3D data into a 1D array is such that the x coordinates increase fastest and the z slowest, each processor should include a range with x*y ghost cells preceding and following its portion of the input data (where x and y are the x and y dimensions of the input), except at the global beginning and end. Also, the `classify_cell` and `isosurface_functor` functors need to subtract the global index of the first element of the local input data from the computed global index before accessing the local input data. If desired, the vertices generated by this algorithm could potentially be welded into a triangle strip as described in a Thrust example [23].

3.2.2 KD-Tree

As shown in Figure 2, the goal of the KD-tree algorithm is to construct a binary tree such that, at each level, the input points belonging to the parent node are evenly distributed to the child nodes based on one of their coordinate values. Sample 2D input points are illustrated in Figure 2A, and the high-level algorithm steps are shown for the first two levels in Figure 2B. First, an initialization step is performed, in which the global rank of each input point is de-

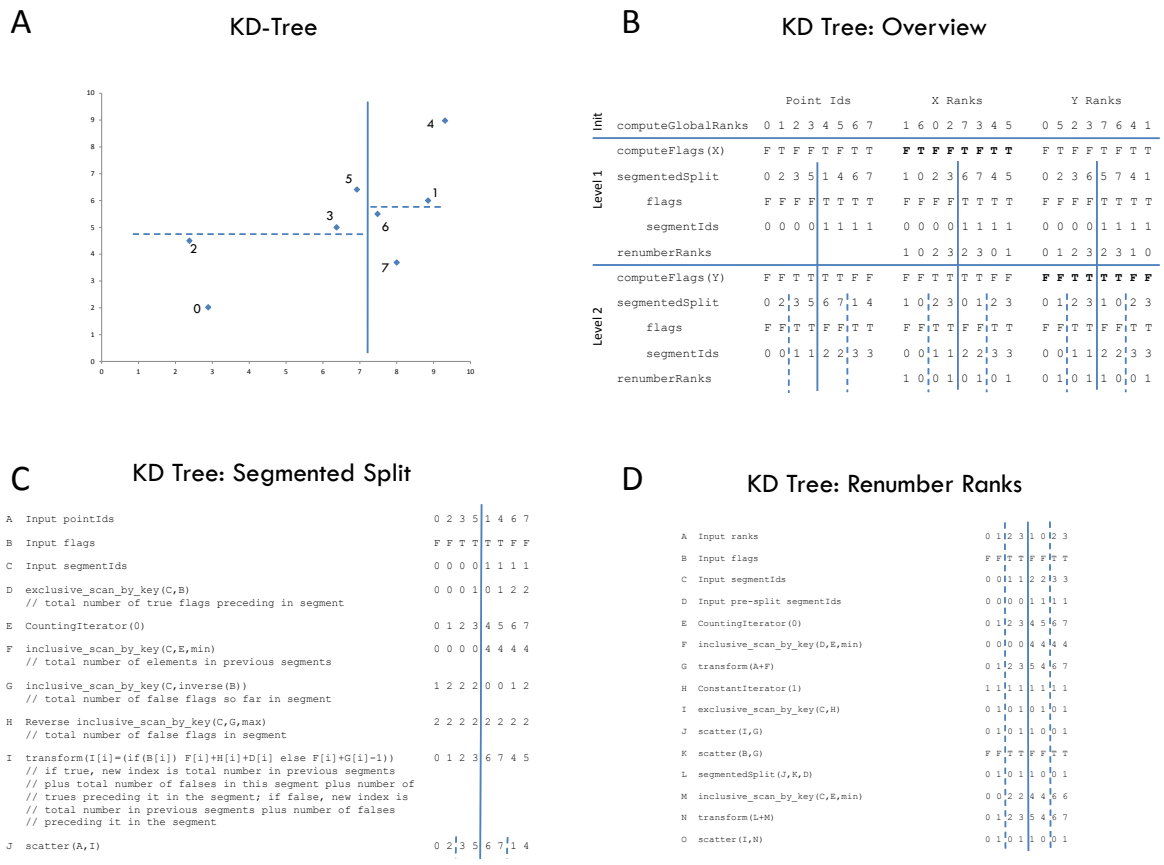

Figure 2: KD-tree algorithm: A) Sample 2D input points, showing two successive levels of splitting (first in x, then in y) B) Overview of the steps in computing the KD-tree for the first two levels C) Details of the segmented split algorithm D) Details of the renumber ranks algorithm

termined for each coordinate dimension. This is accomplished for each dimension by applying the distributed `sort_by_key` operator, where the values are the point ids and the keys the coordinates, and then by scattering a counting iterator according to the sorted values.

At each level, the tree is represented by a segmented vector, where each segment corresponds to a node, and the elements within that segment are the points belonging to the subtree rooted at that node. Thus, there are 2^l segments for the vector at level l. At each successive level, points in one segment are partitioned into two new smaller segments. Since segments represent subtrees, points in a lower level are never in a position outside the range of their parent segment, so global communication is greatest when computing the top levels of the tree and least for lower levels.

At each level, the algorithm then consists of three steps: compute flags (true/false) to mark which points will belong to the left child and which to the right child; split the points to the left and right subtree positions; and convert the points' relative ranks in the parent node in each coordinate dimension to their relative ranks in the child node. In the version of the algorithm we have implemented, the coordinate dimension used for splitting alternates at each successive tree level. The rank renumbering step ensures that, for each segment of length m, we have the ranks 0 through $m-1$ of the points in that segment. Thus, the median point, to be used for the split, is simply the point with rank $m/2$ (with the median itself included in the right subtree for an even number of points). The flags can therefore be generated by first applying a reverse `inclusive_scan_by_key` with the `maximum` operator to a counting iterator to obtain a vector specifying for each element the total number of elements in its segment. A `transform` can then be applied to set the flag for each element depending on whether its rank is less than or greater than or equal to the median rank of its segment (half the total number of elements in the segment). The segmented split step, as illustrated in Figure 2C, computes the new global index for each element, and then uses `scatter` to move each point to its new position in the next tree level. For points with a false flag, the new global index is computed as the total number of elements in previous segments plus the number of falses preceding it in its segment. For points with a true flag, the new global index is computed as the total number of elements in previous segments plus the total number of falses in its segment plus the number of trues preceding it in its segment. The rank renumbering step is somewhat more involved, making use of several scans and scatters, as well as the segmented split function, and is best illustrated with an example as in Figure 2D.

The KD-tree construction algorithm makes extensive use of a number of the distributed primitives, including the segmented versions of `sort` and `scan`. The 3D case, as implemented in our code, is a straightforward extension of the 2D example presented here, with just one more vector for z-coordinate ranks. In most cases, it is likely to incur a fair amount of global communication overhead, particularly in the top levels of the tree. Some of this may be able to be reduced with further refinements of the algorithm. For example, in theory, values should not need to moved outside of

their original segment during the rank renumbering step, since all the necessary information is local to the segment. Nevertheless, this algorithm is a good example of how a type of problem with nested data parallelism can be successfully implemented in this paradigm.

4 RESULTS

Two systems were used for running our performance tests. Up to 128 nodes on the Moonlight supercomputer were used, each having a 16-core 2.6 GHz Intel Xeon E5-2670 CPU, 64 GB of RAM, and an NVIDIA Tesla M2090 GPU. OpenMPI 1.6.3, GCC 4.4.6, and CUDA 5.0 were used. Up to eight nodes on one partition of the Darwin cluster were also used, each having a 48-core 1.9 GHz AMD Opteron 6168 CPU, 128 GB of RAM, and an NVIDIA Quadro 5000 GPU. OpenMPI 1.6.4, GCC 4.4.7, and CUDA 5.0 were used.

Three versions of the isosurface algorithm were compared. The first simply uses our original PISTON isosurface algorithm on a single node. The second is manually configured using MPI to run our original algorithm independently on multiple nodes, with input explicitly divided among the nodes. The third uses our distributed isosurface algorithm, with the input treated as one large vector and the details of the communication hidden in the implementations of the data-parallel primitives. The input was the implicitly-defined "tangle" data set (similar to that used in NVIDIA's Marching Cubes CUDA demo), with equation $(x^4 - 5x^2 + y^4 - 5y^2 + z^4 - 5z^2 + 11.8) * 0.2 + 0.5$). The reported times are the average for the computation only over ten isosurfaces, with the output being vectors (distributed across the nodes) of the vertex positions, normals, and scalar values. They do not include gathering the results back to a root node or rendering images.

Figure 3 and Figure 4 show the performance of these three versions with different input sizes, each run on four nodes using, respectively, the CPU with the Thrust OpenMP backend and the GPU with the Thrust CUDA backend. On the GPU, the distributed algorithm is slower than the manual multi-node version for small data sizes, presumably due largely to the communication overhead (including transfering data between GPU and CPU to be sent to another processor). However, since the amount of communication in this algorithm is on the order of the number of processors rather than the data size, the distributed algorithm converges towards the manual multi-node algorithm for larger data sizes, where both run on four nodes about 3.5 times faster than the single node version.

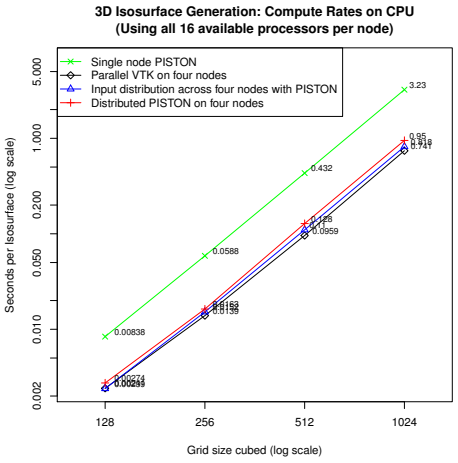

Figure 3: 3D Isosurface Generation: OpenMP Compute Rates on four Intel Xeon E5-2670 nodes on Moonlight supercomputer

As shown in Figure 5, the distributed version run on the CPU makes good use of available on-node parallelism, scaling well with

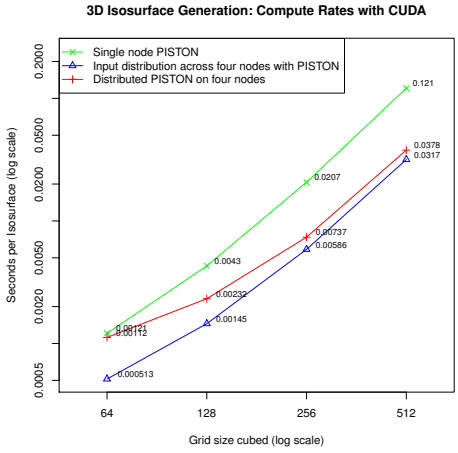

Figure 4: 3D Isosurface Generation: CUDA Compute Rates on four NVIDIA Tesla M2090 GPUs on Moonlight supercomputer

the OpenMP thread count, just as the original single-node version does. Figure 6 shows the scaling with the number of nodes used. Both the distributed version and the manual multi-node version scale well, with just a small roughly constant overhead for the distributed version. The scaling likely benefits from the fact that the input "tangle" field has symmetry properties that result in a relatively good load balance. With different input sets, the load balancing for the final steps that compute the output vertices may be better or worse. However, as long as the percentage of cells generating output is relatively small, the overall time will likely be dominated by the well load-balanced initial steps. In some cases, better performance may perhaps be obtained by calling the `rebalance` operator on `valid_cell_indices` before executing the final steps.

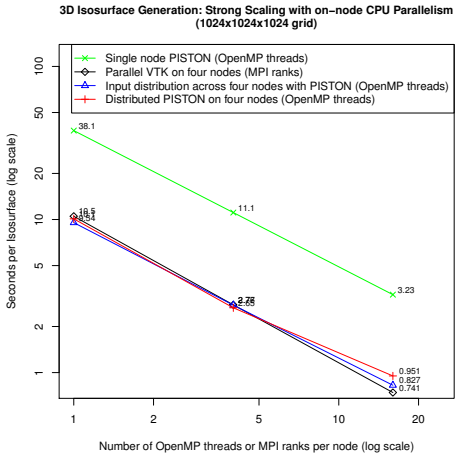

Figure 5: 3D Isosurface Generation: Scaling with on-node CPU parallelism on four Intel Xeon E5-2670 nodes on Moonlight supercomputer

Figures 3, 5, and 6 also compare performance against parallel VTK (based on the ParallelIso example included with VTK, timing only the computation and not rendering). VTK does not use OpenMP or other on-node acceleration constructs, so on-node parallelism for multi-core CPUs is achieved by spawning multiple MPI ranks per node. The scaling performance of parallel VTK is very

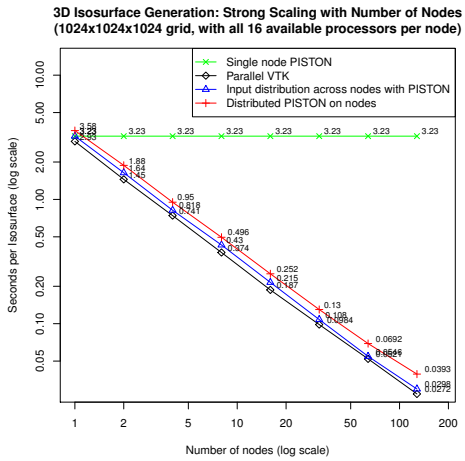

Figure 6: 3D Isosurface Generation: Scaling with the number of Intel Xeon E5-2670 nodes on Moonlight supercomputer

similar to that of distributed PISTON. However, programming VTK to run in parallel requires explicit consideration of the distributed environment, and thus is more complicated than programming serial VTK.

Figure 7 shows that the distributed algorithm compiled using CUDA also scales with the number of GPUs (using one GPU per node), although the parallel efficiency falls off with larger numbers of nodes. This is likely due at least in part to the fact that, with large numbers of GPUs, there is relatively little computational load per GPU. With GPUs, there is also a cost for transfering the data between the GPU and the CPU in order to send it to other nodes, providing less opportunity for gains in computational efficiency to mask the communication overhead. The timings in this figure are with a data size of only 512^3, since larger data sizes will not fit in the memory of the GPUs when distributed among only a few GPUs. However, with a data size of 1024^3, we have observed the distributed algorithm running 1.4 times faster with 64 GPUs than 32 GPUs.

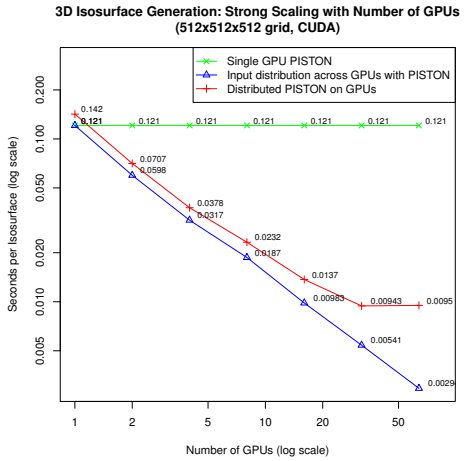

Figure 7: 3D Isosurface Generation: Scaling with the number of NVIDIA Tesla M2090 GPUs on Moonlight supercomputer

However, performance scaling is not the only reason to want to perform a computation distributed across multiple GPUs. In our original PISTON paper, we were not able to compute isosurfaces for a full-resolution 3600x2400x42 ocean temperature data set (from [24]) on a single Quadro 6000 GPU due to memory constraints, so we had to downsample it to an 1800x1200x42 data set. Using distributed PISTON, we are now able to compute isosurfaces on the full-resolution data set across four Quadro 5000 GPUs. An example result rendered from this distributed computation is shown in Figure 8.

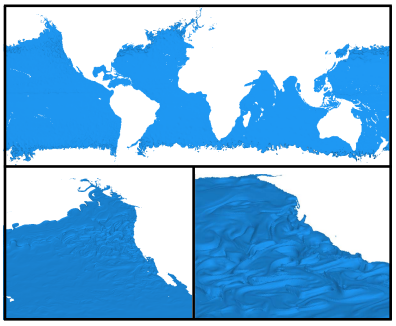

Figure 8: Isosurface at 10^0C computed across four NVIDIA Quadro 5000 GPUs on our Darwin cluster on a 3600x2400x42 ocean temperature data set, showing the level of detail available at this resolution with progressive magnifications along the west coast of North America.

Figure 9 and Figure 10 show the scaling of the distributed KD-tree construction algorithm with the available on-node and inter-node parallelism, respectively. For these tests, the input was a set of 2^{30} randomly generated 3D points. It is able to make good use of up to 16 OpenMP threads on the tested AMD Opteron 6168. The performance also increases with the number of nodes (up to eight, the maximum tested). However, the parallel efficiency decreases significantly as the number of nodes increases. At each successive level of the KD-tree, the number of segments (tree nodes) is doubled, and points are rearranged within their parent segment (i.e., points belonging to the parent node are partitioned into its left and right subtrees). In the first tree level, the points (initially in random order) are partitioned into two subtrees. If there are two processors, on average, half of the points will be moved to the other processor at this level. However, in subsequent levels, the points will only move around within their own processors. If there are four processors, then points will again be moved in the second tree level, but will then all be on their correct final processor. In general, if there are p processors, points may move to a different processor only within the first $log_2(p)$ tree levels. In our experiments, we have confirmed that the time for the first $log_2(p)$ levels increases as p increases (with a large percentage of the time spent in MPI calls), but that the time for each of the remaining $log_2(n) - log2(p)$ levels scales fairly well (with only a tiny fraction spent in MPI calls). Thus, as the number of processors is increased, there is a trade-off between the increased time necessary to move the points to their correct processors in the first $log_2(p)$ levels (plus computing initial global ranks), and the computational savings at subsequent levels. With small data sizes (i.e., a small amount of local work per node), the threshold number of nodes at which scaling stops is smaller than with larger data sizes. Thus, due to the limited amount of memory available on GPUs, we cannot currently scale the KD-tree algorithm well on multiple GPUs.

Overall, the results indicate that algorithms with limited global communication, such as isosurface computation, can be implemented efficiently in the distributed data-parallel programming

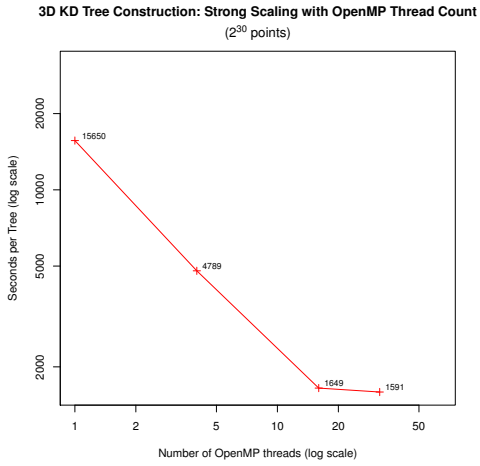

Figure 9: 3D KD Tree Construction: Scaling with the number of OpenMP threads on four AMD Opteron 6168 nodes on Darwin Cluster

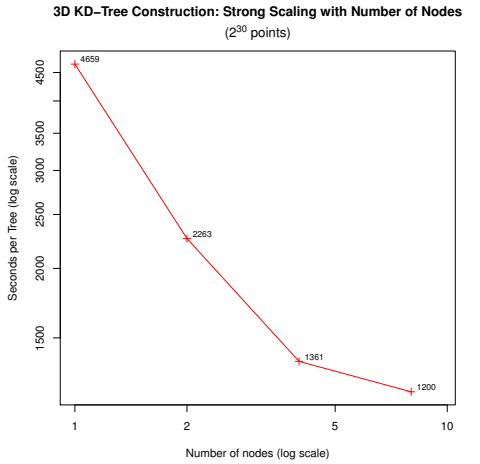

Figure 10: 3D KD Tree Construction: Scaling with the number of AMD Opteron 6168 nodes on Darwin Cluster

model. Those involving significant all-to-all communication, such as KD-tree construction, can also be implemented in this model, but the communication costs will eventually limit the scaling.

5 Conclusion

We have shown how the data parallel programming model we previously demonstrated on multi and many-core architectures on a single node using Thrust can be extended to operate in distributed memory and hybrid distributed-shared memory architectures while providing an almost identical API to the algorithm developer. Algorithmic details and performance results have been provided for two significantly different visualization and analysis algorithms, illustrating that a wide range of algorithms can be implemented in this programming model, including those with nested data-parallelism, with a level of parallel efficiency commensurate with the nature of the problem. Our ongoing and future work is likely to involve further expanding the scope of the data-parallel programming model, extending it to other performance bottlenecks such as simulation computations and disk I/O, in order to provide an integrated end-to-end data-centric programming approach.

Acknowledgements

Aspects of this work were funded by the NNSA ASC CSSE Program; The SciDAC SDAV Institute, funded by the DOE Office of Science through ASCR; and the Los Alamos LDRD Program (XW4W). Phil Jones and Mat Maltrud provided data sets.

References

[1] L. Lo, C. Sewell, and J. Ahrens. PISTON: A Portable Cross-Platform Framework for Data-Parallel Visualization Operators. *Eurographics Symposium on Parallel Graphics and Visualization*, May 2012.

[2] Thrust library: http://thrust.github.com/. Accessed 2013.

[3] W.E. Lorensen and H.E. Cline. Marching Cubes: A High-Resolution 3D Surface Construction Algorithm. *Computer Graphics*, Vol. 21, Num. 4, July 1987.

[4] G. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press. ISBN 0-262-02313-X. 1990.

[5] W.D. Hillis. *The connection machine*. The MIT Press, 1989.

[6] S.L. Johnsson. CMSSL: A scalable scientific software library. *IEEE Scalable Parallel Libraries Conference*, 1993.

[7] Z. Johan, T.J.R. Hughes, K.K. Mathur, S.L. Johnsson. A data parallel finite element method for computational fluid dynamics on the Connection Machine system. *Computer Methods in Applied Mechanics and Engineering*, Vol. 99, Issue 1, Aug. 1992.

[8] L. Bergstrom and J. Reppy. Nested Data-Parallelism on the GPU. *ICFP*, Sep. 2012.

[9] T. Fogal, H. Childs, S. Shankar, J. Kruger, R.D. Bergeron, and P. Hatcher. Large Data Visualization on Distributed Memory Multi-GPU Clusters. *High Performance Graphics*, 2010.

[10] M. Strengert, C. Muller, C. Dachsbacher, and T. Ertl. CUDASA: Compute Unified Device and Systems Architecture. *Eurographics Symposium on Parallel Graphics and Visualization*, 2008.

[11] C. Muller, S. Frey, M. Strengert, C. Dachsbacher, and T. Ertl. A Compute Unified System Architecture for Graphics Clusters Incorporating Data-Locality. *IEEE Trans. on Vis. and Computer Graphics*, 2009.

[12] J. Nieplocha, R.J. Harrison, and R.J. Littlefield. Global Arrays: A Portable Shared-Memory Programming MOdel for Distributed Memory Computers. *ACM/IEEE Conference on Supercomputing*, 1994.

[13] L. Smith and M. Bull. Development of mixed mode MPI / OpenMP applications. *Scientific Programming*, 9 (2001).

[14] T. Peterka, R. Ross, W. Kendall, A. Gyulassy, V. Pascucci, H. Shen, T. Lee and A. Chaudhuri. Scalable Parallel Building Blocks for Custom Data Analysis. *Large Data Analysis and Visualization Sym.*, 2011.

[15] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N.M. Amato, and L. Rauchwerger. STAPL: A Standard Template Adaptive Parallel C++ Library. *Int'l Workshop on Advanced Compiler Technology for High Performance Embedded Systems*, 2001.

[16] K. Moreland, U. Ayachit, B. Geveci, and K. Ma. Dax Toolkit: A Proposed Framework for Data Analysis and Visualization at Extreme Scale. *IEEE Sym. on Large-Scale Data Analysis and Vis.*, Oct. 2011.

[17] J.S. Meredith, S. Ahern, D. Pugmire, R. Sisneros. EAVL: The Extreme-scale Analysis and Visualization Library. *Eurographics Symposium on Parallel Graphics and Visualization*, May 2012.

[18] E. Gabriel et all. Open MPI: Goals, concept, and design of a next generation MPI implementation. *11th European PVM/MPI Users Group Meeting*, 2004.

[19] R.H. Castain, T.S. Woodall, D.J. Daniel, J.M. Squyres, B. Barrett, and G.E. Fagg. The Open Run-Time Environment (OpenRTE): A transparent multicluster environment for high-performance computing. *Future Gen. Comp. Syst.*, 24(2), 2008.

[20] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Comm. of the ACM*, Jan 2008.

[21] H. Shi and J. Schaeffer. Parallel Sorting by Regular Sampling. *Journal of Parallel and Distributed Computing*, 14, 1992.

[22] V. Kale and E. Solomonik. Parallel Sorting Patterns, March 2010.

[23] Thrust examples repository. Accessed 2013. https://github.com/thrust/thrust/blob/master/examples/weld_vertices.cu

[24] M. Maltrud, S. Peacock, and M. Visbeck. On the Possible Long-term Fate of Oil Released in the Deepwater Horizon Incident, Estimated by Ensembles of Dye Release Simulations. *Env. Res. Letters*, 5, 2010.