Spray: Sparse Reductions of Arrays in OpenMP

Jan Hückelheim
Mathematics and Computer Science Division
Argonne National Laboratory
Lemont, IL, USA
https://orcid.org/0000-0003-3479-6361

Johannes Doerfert
Argonne Leadership Computing Facility
Argonne National Laboratory
Lemont, IL, USA
https://orcid.org/0000-0001-7870-8963

***Abstract*—We present Spray, an open-source header-only C++ library for sparse reductions of arrays. Spray is meant for applications in which a large array is collaboratively updated by multiple threads using an associative and commutative operation such as +=. Especially when each thread accesses only parts of the array, Spray can perform significantly better than OpenMP's built-in reduction clause or atomic updates, while also using less memory than the former. Spray provides both an easy-to-use interface that can serve as a drop-in replacement for OpenMP reductions and a selection of reducer objects that accumulate the final result in different thread-safe ways.**

We demonstrate Spray through multiple test cases including the LULESH shock hydrodynamics code and a transpose-matrix-vector multiplication for sparse matrices stored in CSR format. Spray reductions outperform built-in OpenMP reductions consistently, in some cases improving run time and memory overhead by 20X, and even beating domain-specific approaches such as Intel MKL by over 2X in some cases. Furthermore, Spray reductions have a minimal impact on the code base, requiring only a few lines of source code changes. Once in place, Spray reduction schemes can be switched easily, allowing performance portability and tuning opportunities by separating performance-critical implementation details from application code.

***Index Terms*—OpenMP, Reductions, Performance Portability**

I. Introduction

Reductions are omnipresent in high-performance computing. Through the use of an associative and commutative operator, such as summation or multiplication, the dimensionality of a value is "reduced" [1]. For sequential (nonparallel) computers, a reduction of n values is trivially performed by using $n-1$ operations without any additional storage beyond the input and the result. For shared-memory parallel computers, reductions pose a challenge but also provide an opportunity to reduce n values in $log(n)$ steps through a tree-shaped combination. Since the dependencies between the different operations inherit the associative and commutative nature of the individual computations, operations ordered only by *reduction dependencies* [2] can be reordered and even executed in parallel through the use of synchronized accesses or "fix-up" code [3].

This work was funded in part by support from the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357 and the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation's exascale computing imperative. We gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory. We thank Ian Karlin for the helpful discussions about the LULESH test case.

Fig. 1. Spray allows threads to concurrently reduce into overlapping parts of an array while preserving correct results.

Because the reduction primitive is so important, the OpenMP standard has offered a `reduction` clause starting with version 1.0 [4]. The OpenMP standard has extended the support for reduction computations substantially since then, but the underlying principle has not changed: Each thread creates a private copy of the variable or array to which the `reduction` clause is applied. This private copy is used in the associated region instead of the original value, thus avoiding a race condition on the original storage location when partial results are accumulated concurrently. At the end of the associated region all private instances are combined in an implementation-defined order into the original variable or array. This strategy is specified in the OpenMP standard [5], and users may write programs that interact with the private variable copies in almost arbitrary ways while relying on the exact sequence of events described in the standard. It is therefore difficult for OpenMP-compliant implementations to deviate from this strategy without breaking user code.

The strategy prescribed by OpenMP works especially well for scalars or small arrays, but it is only one way of implementing a parallel reduction. In particular, if an application reduces values into a large array and each thread updates only a subset of the elements, then allocating, initializing, and accumulating the entire privatized array for each thread are inefficient. Figure 1 contains a graphical represantation of a domain that is collectively updated by multiple threads, with some overlap between updated areas. In situations like this, programmers often avoid using the OpenMP reduction clause because of its inefficiency, and instead use shared arrays while ensuring safety through atomic updates or synchronisation constructs or by restructuring their computation in application-specific ways. The best strategy in terms of run time and memory footprint depends on the hardware, application, and input data. Since switching between strategies traditionally has involved significant changes to the program code, developers might implement one strategy and leave others unexplored.

In this work we present Spray[1], a header-only library that aims to separate the intent, namely, to *safely accumulate contributions to an array concurrently*, from the implementation. Based on the idea of user-defined reductions [6], Spray has a minimal interface that allows the creation only of "reducer objects," which can be attached to an array, included in an OpenMP reduction clause and then used instead of the original array inside a parallel region. By limiting the interface of Spray reducer objects to the compound assignment operators that are essential for implementing a reduction (e.g., +=), their implementation may internally choose and mix synchronization and privatization strategies without affecting the functional behavior of the program. Every Spray reducer guarantees that all contributions will be visible in the original array at the end of the parallel region. In contrast to the OpenMP-defined reduction operation, Spray leaves all other details of its implementation unspecified. Spray is implemented in a portable way, using only standard C++ and OpenMP features, and has been tested with the most common C++ compilers.

This paper is structured as follows. In Section II we summarize our contributions and limitations and then provide further motivation in Section III. The design and usage of Spray are detailed in Section IV, and Section V describes the Spray reducer objects that we implemented. In Section VII we show evaluation results for an artificial benchmark as well as real-world applications. We conclude in Section IX with a brief summary and an outlook on future work.

II. Contributions and Limitations

The contributions in this paper are

- a clean interface that separates implementation details from the functional specification of a reduction operation,
- configurable reducer objects that are drop-in replacements for the OpenMP reduction clause (as long as the parallel code uses only certain operators allowed by Spray) and that provide different trade-offs between redundant computation, synchronization, and memory overhead, and
- experimental results for a sparse transpose-matrix-vector product, a hydrodynamics code, and a convolution back-propagation kernel, all of which can be easily parallelized by using Spray while offering competitive performance.

[1]Available at `https://github.com/jhueckelheim/spray`

We are aware of the following limitations of our work.

- While the key idea of using interchangeable reduction strategies, or even parts of the Spray implementation, can be used in other shared-memory parallel programming languages, only OpenMP was investigated here.
- We make the same assumptions as OpenMP with regard to associativity of floating-point operations, which may not be suitable if bitwise reproducibility is required.
- Spray relies on operator overloading and the presence of compound assignment operators in C++; a future interface for Fortran or C might be less user-friendly.
- So far, Spray supports only one-dimensional arrays.

III. Motivation

While reductions are often depicted as summations of an array into a scalar variable, they can have many shapes. In practice, not all reductions are dense; that is, not all threads write all reduction locations. A simple sparse reduction is shown in Figure 2. In each iteration, the `out` array is updated by using the associative and commutative addition at two locations. This causes loop-carried dependencies between iterations i and $i+2$ that prevent naive, "DOALL-style" parallel execution of the loop. Various automatic, semi-automatic, and manual approaches for parallelizing such a reduction loop exist. For example, OpenMP provides at least two options: (1) atomically reading and updating the `out` array or (2) privatizing the `out` array together with fix-up code that combines all private copies in the end. Either method removes the loop-carried dependencies without changing the result, at least under the assumption that addition is associative and commutative.

```
for(int i = 1; i < N; ++i) {
  out[i-1] += fn0(in[i]);
  out[i+1] += fn1(in[i]);
}
```

Fig. 2. Sparse reduction with two updates per iteration.

In Figure 3, we illustrate how the `reduction` clause allows sound parallel execution through privatization and fix-up code. As mentioned previously, a conforming OpenMP implementation will allocate, initialize, and accumulate an entire array for each thread participating in the reduction. As the number of threads is increased, each thread is assigned fewer iterations. Given that only two locations are updated per iteration, the number of array elements updated by that thread decreases, while the privatization memory overhead increases. Consequently, each thread will waste increasing amounts of memory, trash the cache, and spend more time operating on unused parts of the private array copies. Furthermore, all compilers we tested allocate the private arrays on the stack, a quality-of-implementation issue that requires the user to modify the OpenMP thread stack size explicitly even for moderately sized arrays, for example, through the `OMP_STACKSIZE` environment variable, or else the program will crash.

Figure 4 shows how atomic updates can be written with OpenMP. In contrast to the previously mentioned scheme,

```
#pragma omp parallel for \
    reduction(+:out[0:N])
for(int i = 1; i < N; ++i) {
  out[i-1] += fn0(in[i]);
  out[i+1] += fn1(in[i]);
}
```

Fig. 3. Concurrent execution of Figure 2 through the reduction clause.

```
#pragma omp parallel for
for(int i = 1; i < N; ++i) {
  #pragma omp atomic update
  out[i-1] += fn0(in[i]);
  #pragma omp atomic update
  out[i+1] += fn1(in[i]);
}
```

Fig. 4. Concurrent execution of Figure 2 through atomic updates.

there is no memory overhead or time wasted initializing or accumulating unused values. However, this scheme is not free of drawbacks either. First, the performance of an atomic update heavily depends on the hardware support for atomic binary operations on the given type. For example, on a system without explicit support for atomic fetch-and-add operations on floating-point values, the atomic update would most likely be implemented with a compare-and-swap (CAS) loop for which the expected performance is substantially lower. In addition to hardware requirements, this scheme puts a burden on the user because all updates need to be annotated in the source code. Doing so is cumbersome and error prone, and it also breaks with best practice for modular software development since a function containing such an update cannot be easily used without inducing overheads on all call sites.

While probably no single best way exists for performing a sparse reduction, the two extremes offered by OpenMP clearly are not always sufficient. In addition to the memory overhead, performance, and usability trade-offs, any fixed solution is likely to suffer from performance-portability limitations—even if memory overhead and performance are good on one system, porting the code to a different system might result in unacceptable memory consumption or run time. Furthermore, reduction locations can depend on dynamic values, as illustrated in Figure 5. The input, here the values in the `col` array, directly impact the performance of the reduction scheme. If two threads are unlikely to access the same location concurrently, atomic updates will perform well. If close-by locations are accessed concurrently, however, privatization is likely faster, since it avoids congestion on contested cache lines. Consequently, a maintainable solution is needed that allows implementing new reduction schemes, as well as switching between schemes statically or dynamically.

```
for(int i = 1; i < N; ++i)
  out[col[i]] += fn(in[i]);
```

Fig. 5. Sparse reductions for which the reduction locations are input dependent.

IV. Spray: Design and Usage

Spray was designed as a flexible high-performance drop-in replacement for sparse OpenMP reductions. To utilize Spray, one has to wrap the original reduction location (the array) into a reducer object, which is used in place of the original reduction location. In a typical workflow, the user chooses the strategy when creating a Spray reducer object; in Figure 6 the `MapReduction` strategy is employed to reduce into an array of `N` double-precision floating-pointer numbers. All but the declaration and use of `sout` match Figure 3. Since any strategy will yield the same result for associative and commutative operations, the user can replace the reduction scheme in a sound way by changing a single line of source code.

```
#include "spray.hpp"

// The reducer object wraps the reduction
// locations out[0:N]. It is templated to
// accept any underlying data type.
spray::MapReduction<double> sout(out, N);

#pragma omp parallel for \
    reduction(+:sout[0:N])
for(int i = 1; i < N; ++i) {
  sout[i-1] += fn0(in[i]);
  sout[i+1] += fn1(in[i]);
}
```

Fig. 6. Sparse reduction using the Spray `MapReduction`.

Figure 7 shows how scoped shadowing can eliminate the need to modify the region associated with the reduction by replacing the original reduction variable with a reducer object of the same name. Regardless of the way Spray is introduced, the modifications to the program are minimal and mechanical. Once Spray is in place, the reduction scheme can be switched easily. For example, to employ atomic updates instead of lazy privatization, one needs only to replace `BlockReduction` in Figure 7 with `AtomicReduction`.

```
#include "spray.hpp"

// Scoped shadowing makes renaming 'out'
// inside the parallel loop unnecessary.
{
 auto &o = out;
 spray::BlockReduction<double> out(o, N);

 #pragma omp parallel for \
     reduction(+:out[0:N])
 for(int i = 1; i < N; ++i) {
   out[i-1] += fn0(in[i]);
   out[i+1] += fn1(in[i]);
 }
}
```

Fig. 7. Sparse reduction using the Spray `BlockReduction`.

The current user interface was deliberately restricted to maximize flexibility while investigating different reduction strategies. Reducer objects could be designed to have a well-defined behavior for more than just the compound assignment operators that we provide, and future versions of SPRAY could offer a systematic way to categorize objects that offer stronger guarantees (for example, a deterministic summation order for bitwise floating-point reproducibility).

We note that the SPRAY reducer objects presented in this work, as well as the built-in OPENMP reductions, do not guarantee reproducible results in the presence of roundoff, even when choosing a static schedule, since the final combination across threads happens in an implementation-defined and possibly nondeterministic order. If a given OPENMP run time consistently uses the same order, then the dense and block-private SPRAY reducer objects will exactly match the summation order of the built-in OpenMP reduction. Additional strategies could be developed with reproducibility in mind.

In principle, SPRAY works with any loop schedule (static, dynamic, guided, etc.) and chunk size, even though we always use the default (static) schedule in our experiments. The schedule and chunk size can affect the performance, since they affect the set of indices accessed by each thread. A small chunk size would probably lead to decreased data locality and hence poor performance in otherwise well-structured problems.

V. SPRAY: REDUCER OBJECTS

SPRAY reducer objects are thin wrappers around existing arrays that provide the necessary operators, for example, `+=`, as well as the OPENMP `declare reduction` construct necessary to use the object in a `reduction` clause. For details on this construct we refer to the OPENMP standard [5].

All SPRAY reducer objects provide an inexpensive constructor that accepts a pointer and a size. While internal data structures might be initialized, no user-facing side effects are performed. The compiler uses the static member function `init`, which is available for each SPRAY reducer object, and initializes the private instance of the reducer object for each thread participating in the reduction. In contrast to the default privatization scheme performed by OPENMP, the `init` method might not allocate significant amounts of (or any) memory. The static member function `reduce` contains the fix-up code that accumulates the intermediate results from one private copy of the reducer object into another copy. As with `init`, the implementation of `reduce` is specific to the object.

SPRAY reducer objects are templated for use with arbitrary types that support the necessary operators. They can therefore be used with arbitrary precision numbers, types that implement reproducible or more accurate summation, platform-specific low-precision types, interval types for error analysis, and so on.

In the following we describe the reducer objects we implemented for our evaluation. A graphical illustration of four implementations is provided in Figure 8. We expect the set of objects to grow over time such that new reduction schemes, perhaps tailored to new hardware features, become available.

a) DenseReduction: The `DenseReduction` in SPRAY mirrors the default reduction scheme described in the OPENMP standard [5]. The entire array is privatized for each thread as part of the `init` method, thus avoiding contention and synchronization. The `reduce` method combines everything at the end through elementwise updates of the entire array. The memory requirement of this scheme grows linearly with the number of threads and size of the original reduction location.

We do not expect this reduction strategy to outperform the default one provided by the compiler, but it still provides advantages. As with all SPRAY reduction objects, one can easily replace the reduction schemes with a single source-line change. Furthermore, the SPRAY `DenseReduction` does, by default, perform heap allocations for the privatized memory, eliminating the need to increase the OPENMP thread stack size via the `OMP_STACKSIZE` environment variable. Moreover, `DenseReduction` is portable and works with user-defined types that provide the appropriate operators, because all SPRAY reducer objects provide a declare reduction construct. Without it, some compilers, most notably `gcc 10.2` and `icc 21.1.9`, do not allow reductions on user-defined types.

b) MapReduction: The `MapReduction` reducer object utilizes one key-value data structure to accumulate intermediate values per thread. This ensures that only updates of the reduction location will cause memory to be allocated and work to be performed. We provide two implementations, one based on the C++ STL `map` type and one using a B-tree.

The first access to a location will insert the key into the map (or B-tree). Since the absence of a key indicates that the location was not previously modified by the thread, there is no need to explicitly initialize privatized copies. Whenever a `MapReduction` object is updated, the relative location into the array is mapped to the new value of the privatized element. At the end of the parallel region, maps are merged elementwise and eventually added to the original array.

By using standard containers to store the incoming updates, the implementation is straightforward using either C++ STL maps or B-trees. While the latter outperformed the former, neither performed, partly because they provide additional functionality that is not needed for a reducer object.

c) AtomicReduction: The `AtomicReduction` atomically updates the original storage locations. This corresponds to the idea illustrated in Figure 4 but without the need to annotate the actual source accesses. Instead, the `AtomicReduction` utilizes the `#pragma omp atomic update` inside the overloaded binary operators. The `AtomicReduction` reducer object that is created for each thread needs to know only the location of the original storage locations in order to access it directly through atomic updates. Neither the `init` nor the `reduce` method performs any additional work, and no dynamic memory allocations are required.

The `AtomicReduction` reducer object is the best choice to avoid memory overhead. It performs well if the threads mostly update different locations mapping to distinct cache lines. Without contention on memory locations, this scheme still induces a higher latency on the update, which, depending on the system and surrounding code, may or may not cause overheads. If atomic accesses are expensive or there is contention, this scheme might not scale well.

d) BlockReduction: The `BlockReduction` privatizes the original locations lazily by dividing the array into statically sized blocks that are then privatized individually on demand. The `init` method will allocate and initialize only internal management data structures, but not privatized memory. Whenever an update is performed on a nonprivatized location, a block is assigned in which intermediate values are accumulated. In the `reduce` method the blocks are merged elementwise.

We provide two distinct flavors of the `BlockReduction` reducer object. The first one ("block-private") creates privatized blocks whenever an access is made to a location for which the thread does not yet have a block. This strategy will result in the same summation order as the dense strategy, since the only difference lies in the treatment of unused elements. The second allows threads to own blocks in the original array for direct access. To ensure soundness for this second scheme, the threads must communicate ownership of array blocks and use privatization as a fallback. We provide an implementation with OPENMP locks ("block-lock") as well as one using C++ atomic compare-and-swap operations ("block-CAS").

The block size provides a hyperparameter to balance the number of block allocations with the overhead through work on unused parts within blocks. A wide range of values performed well in our experiments. We refer to a reducer object with a certain block size by appending the size to the name, such as ("block-CAS-1024").

e) KeeperReduction: The `KeeperReduction` distributes ownership of the reduction locations statically across threads. Updates of locations owned by the thread are performed nonatomically on the original storage. If the location is not owned by the thread, an "update request" is created and enqueued with the location owner. These requests describe the location as well as the update value. At the end of the region associated with the reduction all update requests are applied concurrently to the original source locations by the respective owner. Other ownership models could be investigated to make the `KeeperReduction` performant in more situations.

VI. Test Cases

We evaluate the memory footprint and run-time performance of SPRAY in three test cases: a convolution back-propagation kernel in Section VI-A, a transpose-matrix-vector multiplication in Section VI-B, and a shock hydrodynamics code in Section VI-C. First, we describe the testing methodology.

All tests were run on an Intel Xeon Platinum 8180M CPU clocked at 2.50 GHz with 1 MB L2 cache per core and 27.5 MB L3 cache, referred to as *Skylake*. The system is actually a dual-socket machine with two Skylake CPUs, but we used only one socket with 28 physical cores and up to 56 threads, to avoid NUMA issues during our measurements. The entire system has 384 GB of RAM.

For the back-propagation test case in Section VI-A, we used the Intel icc compiler version 19.0.4.243 with compiler flags `-qopenmp -xHost` and both the GNU compiler version 9.2.0 and LLVM (clang) compiler version 11.0 with flags `-fopenmp -march=native`. We report results for all three compilers at the optimization levels `O1`, `O2`, and `O3`. For the other two test cases, we performed measurements only with the Intel icc compiler and optimization level `O2`. The aim was to reduce experimentation time; the test cases can be run with any of the three compilers or optimization settings.

All runs were configured to take at least multiple seconds by adjusting problem sizes and/or iteration counts, in order to reduce measurement noise. Additionally, run-time experiments were repeated at least 10 times, and we report mean results. For the back-propagation and transpose-matrix-vector product test cases we used the Google benchmark suite to automatically adjust iteration counts and collect run-time statistics. The shock hydrodynamics code is a standalone application and more challenging to embed into the Google benchmark suite. We therefore used our own script to repeat time measurements.

Memory footprints were measured by using the GNU time utility, which can report the *maximum resident set size* throughout a program's run time. The reported numbers include memory reserved for the program's stack, heap, the executable itself, and potentially also shared libraries. We report only the *overhead* of parallel reductions, which we calculate by

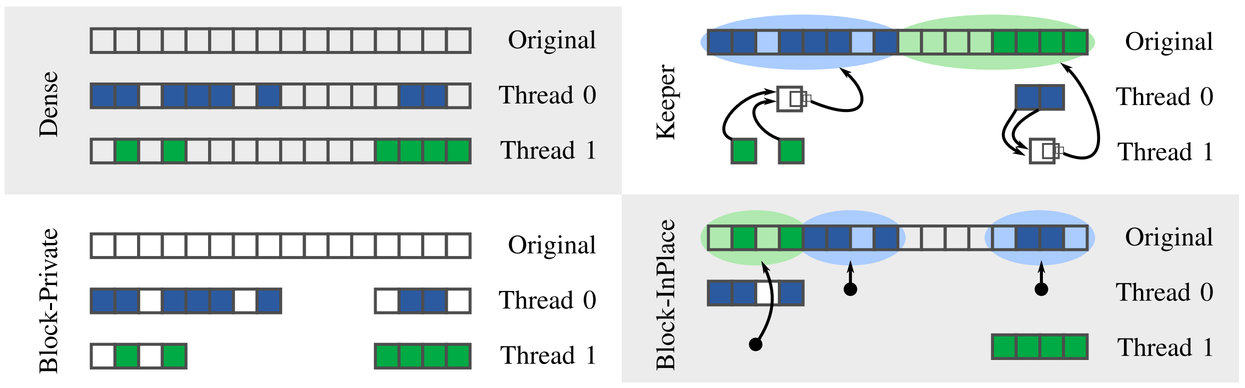

Fig. 8. Illustration of a reduction with two threads using four different SPRAY reducer object implementations. Each box represents a memory location; the boxes for Thread 0 and Thread 1 are privatized locations. The locations updated by Thread 0 and Thread 1 are depicted as blue and green boxes, respectively.

subtracting the measured resident set size of the sequential program from that of the corresponding parallel program. We found that the measurements fluctuate about 5 MB between runs of the same program with identical inputs, occasionally causing negative calculated overheads for parallelization approaches that do not require much additional memory.

A. 1D Convolution Back-Propagation

Convolutions are a frequently used motif in scientific computing, machine learning, image processing, and other areas. They can be implemented as a stencil loop in which an index in an output array (or tensor) is updated by a function that gathers data from a neighborhood in an input array (or tensor). These loops are trivially parallel. Back-propagation or automatic differentiation in reverse mode through such a computation results in a loop in which contributions are scattered to a neighborhood around the loop index, as shown in Figure 9. Such loops cannot be parallelized naively [7], [8].

```
for(int i = 1; i < N-1; ++i) {
  out[i-1] += wl * in[i];
  out[i]   += wc * in[i];
  out[i+1] += wr * in[i];
}
```

Fig. 9. Sparse reduction back-propagating through a 3-point stencil.

While loop transformation techniques might be able to remove or mitigate the effects of the loop-carried reduction dependencies [9], [10], SPRAY makes performant parallelization easy for the user. We discuss the results of OPENMP and SPRAY reductions on this example with an array of 10^7 single-precision floating-point numbers in Section VII-A.

B. CSR Transpose-Matrix-Vector Product

Matrix-vector products and transpose-matrix-vector products are some of the most frequently used operations in scientific computing. Common formats to store sparse matrices (matrices that contain relatively few nonzero entries) include the compressed sparse row (CSR) and compressed sparse column (CSC) format. These formats are used not only for storing such matrices in files but also for holding these matrices in memory while using them in computations. Because of the analogy between (sparse) matrices and graphs (that are not fully connected), algorithms that operate on matrices can often be represented as graph algorithms, and vice versa. For example, the PageRank algorithm in [11] is based on the CSR format and similar to the repeated application of matrix-vector-products. In addition to showing performance on linear algebra problems, this test case can therefore be seen as a proxy for sparse reductions that occur in graph problems.

A straightforward implementation for performing transpose-matrix vector products on CSR matrices (or performing matrix-vector products on CSC matrices) contains updates to data-dependent output locations, as illustrated in Figure 10.

This loop can easily be parallelized as a reduction to the output array `res`. In Section VII-B we discuss the performance of

```
for(i = 0; i < num_rows; ++i)
  for(k = rwptr[i]; k < rwptr[i+1]; ++k)
    res[cols[k]] += vals[k]*x[i];
```

Fig. 10. Sparse reduction computing a CSR transpose-matrix-vector product.

SPRAY and OPENMP reductions in this case. We compare this with the corresponding functions provided by the Intel math kernel library (MKL) [12], namely, `mkl_cspblas_scsrgemv` (a legacy one-call subroutine), and an inspector/executor-based method that operates on an optimized internal representation of the matrix (`mkl_sparse_s_mv`).

We apply these implementations to two matrices: the `s3dkt3m2` matrix from the Matrix Market [13], with 90k rows/columns and 1.9M nonzero entries, and the `debr` matrix from the University of Florida Sparse Matrix Collection [14], a 1M by 1M matrix with 4M nonzero entries. While the matrices are actually symmetric, this information was not explicitly made available to the implementations, and all operations were performed as if applied to general matrices.

C. LULESH Shock Hydrodynamics

The Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) code is a widely studied proxy application that models the hydrodynamics equations. It has been ported to a variety of architectures and programming models [15]. We used the OpenMP-only configuration of LULESH 2.0 [16] and manually adjusted the problem size to $90 \times 90 \times 90$ with 100 iterations.

The functions `IntegrateStressForElems` and `CalcFBHourglassForceForElems` contain sweeps over an unstructured mesh (or graph), with updates to data-dependent array indices that prevent naive “DOALL-style” parallelization. A domain-specific parallelization approach is activated when the code detects more than one available thread, which requires that the output array be replicated 8 times. Updates are then organized such that no more than one thread accesses a given instance of the output array. The instances are combined in an additional sweep over the mesh. We removed this scheme, greatly simplifying the affected functions, and instead used SPRAY reductions to safely accumulate contributions to the output array. We discuss the performance implications of this approach in Section VII-C.

Note that we report the entire run time as printed by the LULESH code, not just the run time of the parallel loops that we modified. This means that the run-time differences between the parallelization approaches are relatively small, since other parts of the code that contribute significantly to the overall time were left unchanged.

VII. RESULTS

We make some general remarks about the performance of SPRAY and OPENMP reductions here, before discussing the test cases in more detail in the following subsections.

In our tests, dense reductions, whether implemented in SPRAY or compiler provided, perform poorly, often worse than sequential code. Increasing the thread counts degrades performance further because more threads allocate, initialize, and accumulate private locations while fewer are actually updated per thread.

Atomic updates appear to have a constant time overhead over nonatomic memory accesses but scale well in our experiments and can outperform other implementations for large thread counts. The performance of SPRAY atomic reducer objects relies on the compiler's ability to eliminate the abstraction layer. This works well on the Intel icc compiler, whereas LLVM and GNU compilers have 5–10% run-time overhead when using SPRAY atomics compared with OPENMP atomics.

The keeper and block-based SPRAY reducer objects are consistently among the fastest implementations, outperforming even some of the domain-specific parallelization schemes in the Intel MKL and LULESH. Map-based reductions were not competitive and are not included in the remaining discussion.

Overall, atomics are useful for avoiding memory overhead and where reduction accesses are few and without contention. Block-based reducers perform best when reduction accesses have high locality, both temporal and spatial. The keeper reduction excels if the updated indices on each thread closely match the static ownership structure.

A. 1D Convolution Back-Propagation

Figure 11 shows the scalability results for the 1D convolution back-propagation (ref. Section VI-A). The use of atomic updates outperforms the sequential code with 8 or more threads. This result is not surprising, since the problem structure makes memory conflicts during atomic updates unlikely. Most SPRAY reducer objects have a lower overhead than the atomic updates have and also scale well, offering absolute speedups compared with the sequential code if two or more threads are used. Again, the problem structure causes almost no overlap of the update locations accessed by different threads, and updates within any thread occur on nearby array indices, which explains why block-based approaches work well. Furthermore, the near-perfect one-to-one mapping between the loop counter and array update location matches the ownership model in the keeper implementation, which explains its good performance.

Figure 12 shows that the compiler and optimization level affect the performance of SPRAY and OPENMP alike. Nevertheless, SPRAY consistently outperforms built-in OPENMP reductions in this test. The best performance overall, at 1.38 s per 1,000 iterations, was obtained with icc. Clang managed 1.49 s, while gcc peaked at 2.28 s per 1,000 iterations. The best sequential time was 21.56 s per 1,000 iterations with icc.

Figure 13 shows a variety of SPRAY implementations and different block sizes. For this test case, larger block sizes are almost always better than smaller ones. Some reducer objects scale more poorly on gcc, and very small block sizes do not scale well with any compiler.

B. CSR Transpose-Matrix-Vector Product

We used two CSR matrices in a transpose-matrix-vector product as described in Section VI-B. The results for the `s3dkt3m2` input are shown in Figure 14 and for the `debr` input in Figure 15. We omitted the noncompetitive configuration in which the time measurement included the time taken by the MKL inspector to optimize the matrices.

The `s3dkt3m2` matrix has a narrow bandwidth (i.e., it is almost diagonal), and with 7 MB it is small enough to fit in cache, even with 56 threads and a dense reduction scheme. (Note that the reduction needs to replicate only the result vector, not the matrix.) Consequently, we see good scaling behavior even in this case. Still, with high thread counts the best reducer objects are block-based, and the block-CAS object is fastest with 179.062 ms. The single-call MKL function (`mkl_cspblas_scsrgemv`, MKL legacy) performs reasonably for low thread counts but shows poor scaling and peaks at 542.828 ms with 4 threads. The inspector/executor function (`mkl_sparse_s_mv`, MKL I/E) without additional hints performs better than the legacy version but nevertheless peaks at 8 threads, with a time of 275.848 ms.

The fastest version is the inspector/executor function with hints about the nature of the upcoming operations provided prior to the multiplication, taking 25.818 ms. While this configuration performs best, it is competitive only because the hint-based optimizations are not included in the time. If they are included, it is slower than the sequential code. The memory overhead of the MKL I/E with hints is far greater than that of the dense reductions. Block-based schemes and MKL functions without hint have a significantly smaller memory footprint.

The `debr` matrix also has structure but is bigger and has a broader bandwidth. Since the data does not fit into cache any more, it is not surprising that dense approaches perform poorly. The results are otherwise similar to those of the `s3dkt3m2` case except that the increased pressure on the memory subsystem allows atomic updates (taking 502.053 ms) to surpass block-lock (at 548.413 ms) for 56 threads. On 28 threads, block-lock is still better than atomics at 685.502 ms vs 856.849 ms.

This test case shows that both the input and platform-dependent properties can impact the reduction performance. The fact that task-specific APIs like the legacy MKL are not immune to this highlights the need for a simple way to adapt the reduction scheme in order to achieve performance portability.

C. LULESH Shock Hydrodynamics

Just as in the other test cases, dense reductions become slow when the number of threads is increased beyond a certain number, in this case 16. However, dense reductions are the best choice, or close to it, for up to 4 threads. The atomic reducer object is competitive for high thread numbers (28 and 56), while block-based reducers as well as the domain-specific original approach perform well across the entire range of thread counts. The SPRAY block-lock reducer outperforms all others, taking 4.68 s and 4.63 s, respectively, on 28 and 56 threads. The fastest run time for the original implementation, achieved with 28 threads, is 5.25 s.

The memory footprint of dense reductions grows quickly, to over 4 GB on 56 threads, despite this being a small problem

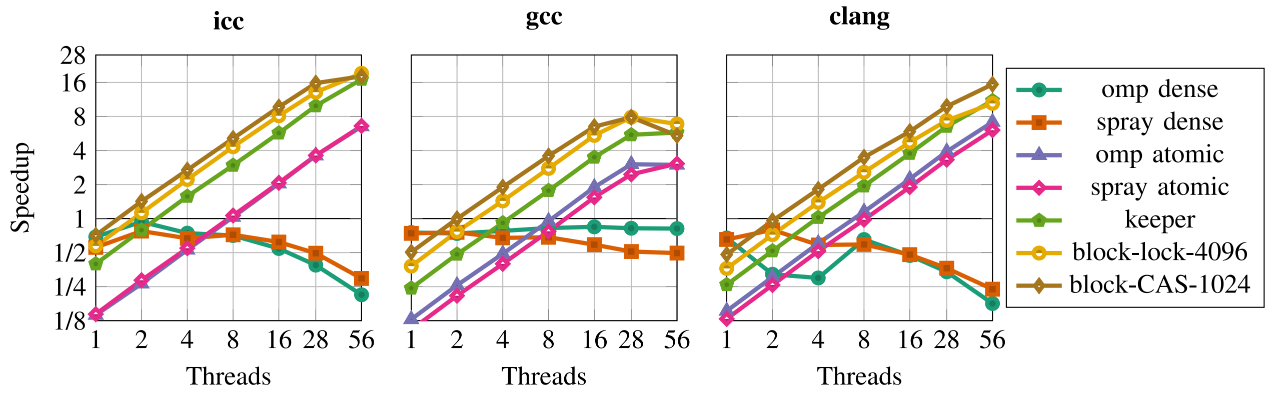

Fig. 11. Speedup of built-in OpenMP and selected Spray reduction implementations over the sequential convolution back-propagation, on Intel, GNU, and LLVM compilers. All parallel implementations have some overhead. Hence, their lines start below 1, indicating a slowdown, and rise above 1, indicating a speedup as more threads are added. The sequential version is created with each compiler; hence the baseline performance differs between subplots, and better scalability with one compiler does not imply better absolute run time.

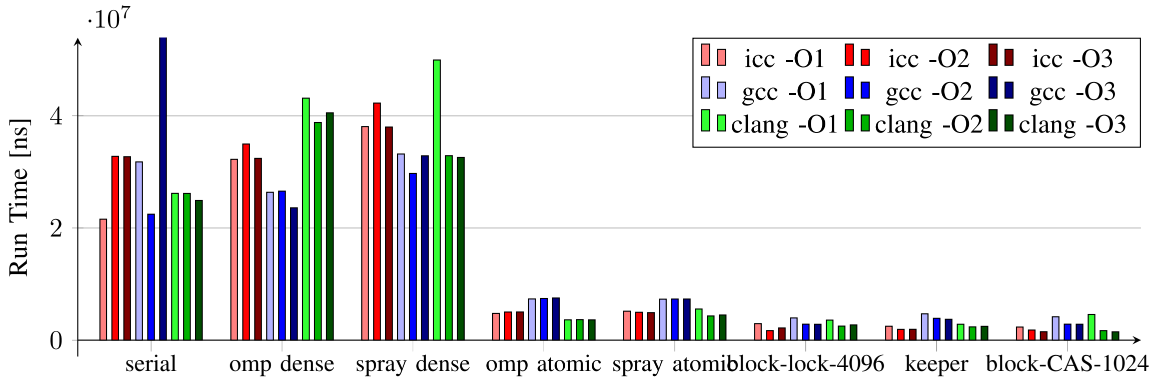

Fig. 12. Absolute run times per iteration (best across all tested thread counts) for various compiler and optimization settings applied to the convolution back-propagation. The best performance was often achieved with 56 threads, except for dense reductions, which usually performed best with 2 threads (but still worse than the sequential code). The GNU compilers tend to do well with sequential code and dense reductions and perform more poorly than other compilers for code containing atomics.

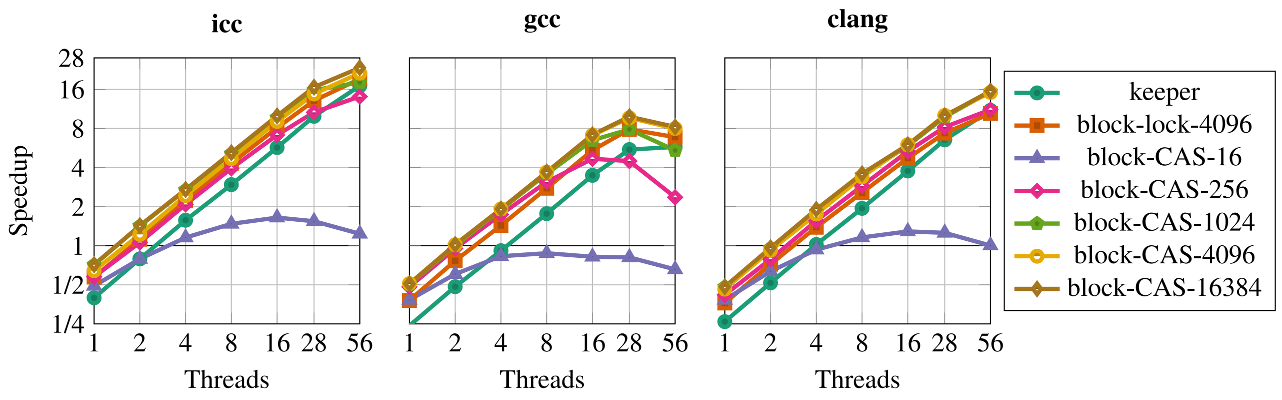

Fig. 13. Scalability of various Spray backends and parameter settings over the sequential convolution back-propagation. Keeper, block-lock, and block-CAS with block sizes above 256 perform well across all tested compilers. The performance differences between different block sizes with good scalability in this plot can still be significant. For example, block size 1,024 on icc gives an absolute run time of 1805422.821 s, compared with 1533950.73 and 1395198.21 for block sizes of 4,096 and 16,384—a decrease of 15% and 32%, respectively, compared with the 1,024 block size.

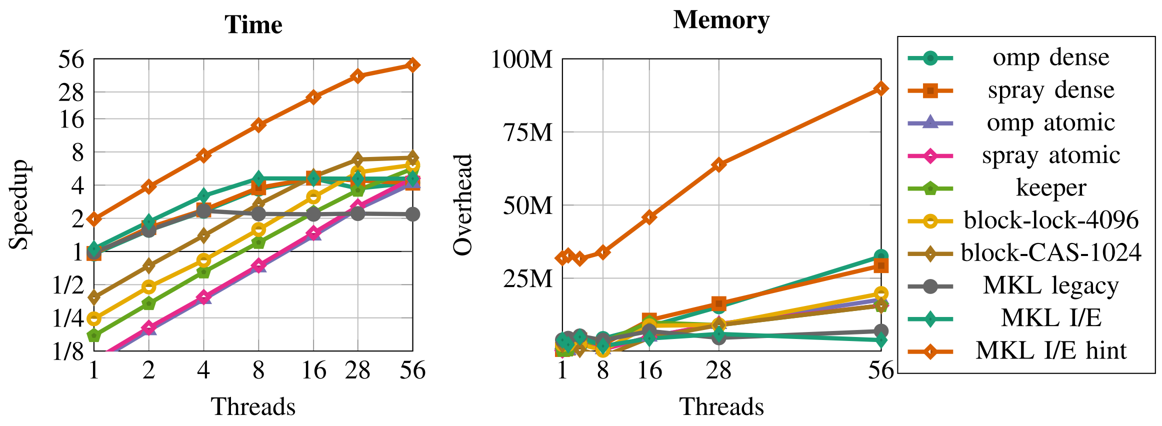

Fig. 14. **Left:** Scalability for transpose-matrix-vector products on the S3DKT3M2 matrix. The MKL inspector/executor implementation can be seen to perform best but has an unfair advantage since it can analyze the matrix before the time measurement starts. Without this advantage, it can no longer outperform the block-lock or atomic reducer objects on 56 threads. Dense reductions yield competitive performance up to 8 threads, but implementations using locks perform better beyond that. **Right:** The memory footprint of the MKL inspector/executor with hint far exceeds that of all other approaches. With dense reductions, the memory footprint grows with the number of threads but remains below the size of the L3 cache of our test machine.

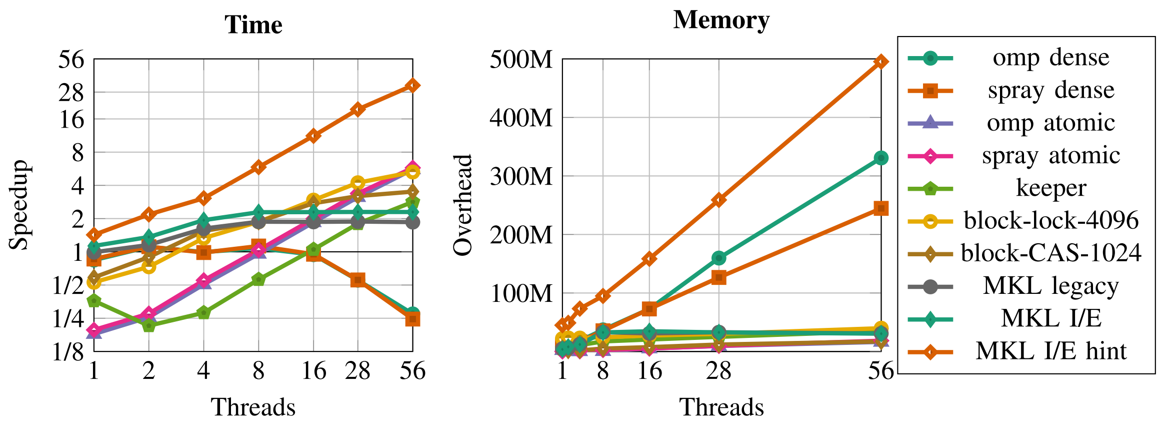

Fig. 15. **Left:** Scalability for transpose-matrix-vector products on the debr matrix. Just as in Figure 14, the MKL inspector/executor implementation performs very well when allowed to analyze the matrix beforehand. Otherwise, some SPRAY reducers outperform both the legacy and the new inspector/executor-based MKL implementation on 56 threads by a factor of 2.8 and 2.3, respectively. **Right:** The memory footprint of the MKL inspector/executor with hint again exceeds all others. Dense reductions also use significant amounts of memory, and built-in OPENMP reductions appear to use more than SPRAY dense reductions.

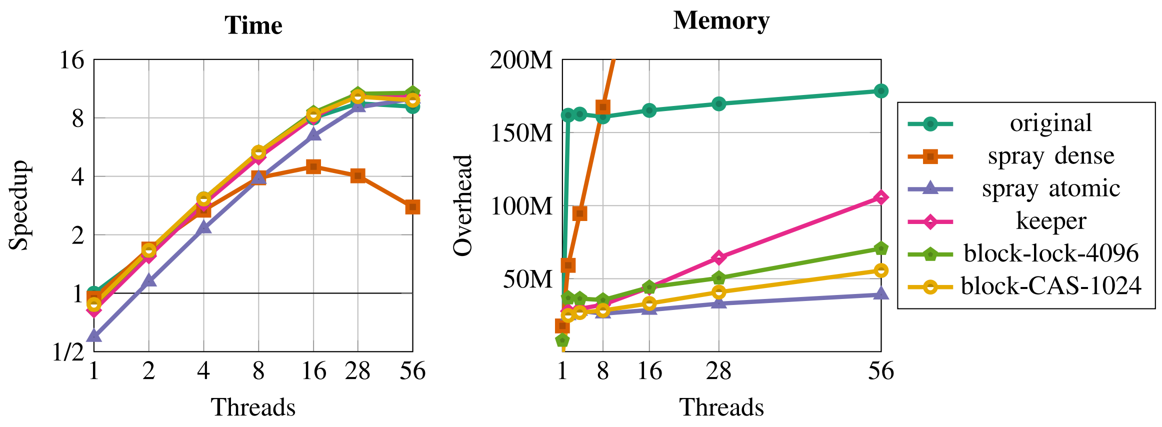

Fig. 16. **Left:** Scalability of the LULESH code. With the exception of dense reductions, all implementations scale well, although atomics have a constant-factor overhead and become competitive only at high thread counts. The SPRAY block-lock reducer outperforms the domain-specific parallelization scheme in LULESH by over 10% on 56 threads. **Right:** The memory footprint of dense reductions grows rapidly, up to 4 GB for this relatively small test case (the plot truncates this at 200 MB). The domain-specific scheme in LULESH has a memory overhead that is larger than that of any nondense SPRAY reducer.

size. Because the original parallelization creates 8 copies when executed multithreaded, it has a memory footprint that jumps when going from 1 thread to 2 and stays more or less constant beyond that. All SPRAY reducer objects tested here except for the dense one actually use less memory than the original scheme does, while providing similar or better performance without the need to handle the synchronization explicitly.

VIII. Related Work

Reductions have been studied for decades, and the body of related work is consequently large. We discuss closely related work and provide pointers for further reading.

SPRAY provides simple-to-use, performant, and exchangeable reduction schemes, made possible in native C++ with OPENMP because of user-defined reductions [6].

Shirako et al. proposed and evaluated three reduction schemes for tasks in the Habanero–Java environment [17]. Also, the OmpSs run time has been extended with various reduction schemes [18], [19]. While we share similar ideas, only SPRAY provides flexible and straightforward integration into existing OPENMP codes. The reducer object choices are more elaborate and completely implemented in source code, thus allowing portability between compilers and extensibility in the future. In addition, we provide an extensive evaluation of memory overhead and performance on real-world problems.

An early comparison of different schemes for sparse reductions was presented by Han and Tseng [20]. The dissertation by Ciesko [21] provides an extensive and more up-to-date background and related work chapters. Similarly, the dissertation by Doerfert [22] summarizes reduction detection, modeling, and optimization with a focus on polyhedral approaches.

IX. Conclusion and Outlook

SPRAY provides a novel collection of reducer objects that can be used through an easy-to-use interface, enabling performance portability and tuning opportunities with minimal code modifications. Our evaluation of SPRAY clearly shows that SPRAY is superior to other OPENMP solutions, and even application-specific implementations, with regard to performance, memory overhead, and usability.

We expect future work not only to add more reducer objects to capitalize on hardware features and domain knowledge but also to provide a generic reducer object that moves the burden of picking a strategy from the user to the compiler and run time. In addition, GPU support and the ability to natively handle multidimensional arrays is under investigation.

References

[1] S. P. Midkiff, *Automatic Parallelization: An Overview of Fundamental Compiler Techniques*, ser. Synthesis Lectures on Comp. Arch., 2012.

[2] W. Pugh and D. Wonnacott, "Static Analysis of Upper and Lower Bounds on Dependences and Parallelism," *Trans. Program. Lang. Syst.*, 1994.

[3] W. M. Pottenger and R. Eigenmann, "Idiom Recognition in the Polaris Parallelizing Compiler," in *Proceedings of the 9th international conference on Supercomputing, ICS*, 1995.

[4] OpenMP Architecture Review Board, "OpenMP C and C++ Application Program Interface – Version 1.0," www.openmp.org/wp-content/uploads/cspec10.pdf, 1989.

[5] O. A. R. Board, "OpenMP Application Program Interface 5.0," www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf, 2018.

[6] A. Duran, R. Ferrer, M. Klemm, B. R. de Supinski, and E. Ayguadé, "A Proposal for User-Defined Reductions in OpenMP," ser. Lecture Notes in Computer Science, vol. 6132, 2010. [Online]. Available: https://doi.org/10.1007/978-3-642-13217-9_4

[7] T.-M. Li, M. Gharbi, A. Adams, F. Durand, and J. Ragan-Kelley, "Differentiable Programming for Image Processing and Deep Learning in Halide," *ACM Transactions on Graphics (TOG)*, vol. 37, no. 4, 2018.

[8] J. Hückelheim, P. Hovland, M. Strout, and J.-D. Müller, "Parallelizable Adjoint Stencil Computations Using Transposed Forward-Mode Algorithmic Differentiation," *Optimization Methods and Software*, no. 4-6, 2018.

[9] K. Stock, M. Kong, T. Grosser, L.-N. Pouchet, F. Rastello, J. Ramanujam, and P. Sadayappan, "A Framework for Enhancing Data Reuse via Associative Reordering," in *ACM SIGPLAN Notices*, vol. 49, no. 6. ACM, 2014.

[10] J. Doerfert, K. Streit, S. Hack, and Z. Benaissa, "Polly's Polyhedral Scheduling in the Presence of Reductions," *CoRR*, vol. abs/1505.07716, 2015. [Online]. Available: http://arxiv.org/abs/1505.07716

[11] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP Benchmark Suite," *CoRR*, vol. abs/1508.03619, 2015. [Online]. Available: http://arxiv.org/abs/1508.03619

[12] *Intel Math Kernel Library. Reference Manual*. Intel Corporation, 2009.

[13] R. F. Boisvert, R. Pozo, K. Remington, R. F. Barrett, and J. J. Dongarra, "Matrix Market: A Web Resource for Test Matrix Collections," in *Quality of Numerical Software*. Springer, 1997, pp. 125–137.

[14] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, 2011.

[15] I. Karlin, et al., "LULESH programming model and performance ports overview," Tech. Rep. LLNL-TR-608824, December 2012.

[16] I. Karlin, J. Keasler, and R. Neely, "Lulesh 2.0 updates and changes," Tech. Rep. LLNL-TR-641973, August 2013.

[17] J. Shirako, V. Cavé, J. Zhao, and V. Sarkar, "Finish accumulators: An efficient reduction construct for dynamic task parallelism," in *Languages and Compilers for Parallel Computing LCPC*, 2012.

[18] J. Ciesko, J. Bueno, N. Puzovic, A. Ramírez, R. M. Badia, and J. Labarta, "Programmable and Scalable Reductions on Clusters," in *27th IEEE International Symposium on Parallel and Distributed Processing, IPDPS*, 2013. [Online]. Available: https://doi.org/10.1109/IPDPS.2013.63

[19] J. Ciesko, S. Mateo, X. Teruel, X. Martorell, E. Ayguadé, and J. Labarta, "Supporting Adaptive Privatization Techniques for Irregular Array Reductions in Task-Parallel Programming Models," in *12th International Workshop on OpenMP, IWOMP*, ser. Lecture Notes in Computer Science, vol. 9903, 2016. [Online]. Available: https://doi.org/10.1007/978-3-319-45550-1_24

[20] H. Han and C. Tseng, "A Comparison of Parallelization Techniques for Irregular Reductions," in *Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS-01)*, 2001. [Online]. Available: https://doi.org/10.1109/IPDPS.2001.924963

[21] J. Ciesko, "On algorithmic reductions in task-parallel programming models," Ph.D. dissertation, 2017.

[22] J. Doerfert, "Applicable and sound polyhedral optimization of low-level programs," Ph.D. dissertation, Saarland University, Saarbrücken, Germany, 2018. [Online]. Available: https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/28318