Optimizing the LULESH Stencil Code using Concurrent Collections

Chenyang Liu
Purdue University
465 Northwestern Avenue
West Lafayette, Indiana
Liu441@Purdue.edu

Milind Kulkarni
Purdue University
465 Northwestern Avenue
West Lafayette, Indiana
Milind@purdue.edu

ABSTRACT

Writing scientific applications for modern multicore machines is a challenging task. There are a myriad of hardware solutions available for many different target applications, each having their own advantages and trade-offs. An attractive approach is Concurrent Collections (CnC), which provides a programming model that separates the concerns of the application expert from the performance expert. CnC uses a data and control flow model paired with philosophies from previous data-flow programming models and tuple-space influences. By following the CnC programming paradigm, the runtime will seamlessly exploit available parallelism regardless of the platform; however, there are limitations to its effectiveness depending on the algorithm. In this paper, we explore ways to optimize the performance of the proxy application, Livermore Unstructured Lagrange Explicit Shock Hydrodynamics (LULESH), written using Concurrent Collections. The LULESH algorithm is expressed as a minimally-constrained set of partially-ordered operations with explicit dependencies. However, performance is plagued by scheduling overhead and synchronization costs caused by the fine granularity of computation steps. In LULESH and similar stencil-codes, we show that an algorithmic CnC program can be tuned by coalescing CnC elements through step fusion and tiling to become a well-tuned and scalable application running on multi-core systems. With these optimizations, we achieve up to 38x speedup over the original implementation with good scalability for up to 48 processor machines.

Keywords

Concurrent Collections, LULESH, Tiling

1. INTRODUCTION

Effectively programming scientific applications is a very challenging task. There are many research fields dedicated to producing new models and methods for simulating real world phenomenon. However, the domain scientists who develop these models require expertise in parallel programming to effectively run their applications in order to take advantage of modern and future architectures. An intuitive way to solve this problem is to use an architecture-agnostic programming language that separates the responsibilities of the domain scientist from the programming expert. This can be achieved using Concurrent Collections (CnC), a parallel programming model that combines ideas from previous work in tuple-spaces, streaming languages, and dataflow languages [2, 7, 13].

In CnC, algorithmic programs are expressed as a set of partially-ordered operations with explicitly defined dependencies. The CnC runtime exploits potential parallelism, so long as it satisfies the dependencies set by the programmer. The responsibility of the programmer is minimal, only requiring them to provide the basic semantic scheduling constraints of their problem. The advantages of simplified programmability comes at the expense of sub-optimal fine-grained parallelism for these programs. In this paper, we explore how to rewrite CnC programs to make specific equivalent-state transformations to boost parallel performance. We investigate the effects of these transformation techniques on the Livermore Unstructured Lagrange Explicit Shock Hydro (LULESH) mini-app, a hydrodynamics code created in the DARPA UHPC program [9, 11].

We discover that there are many opportunities to tune complex CnC programs. Algorithms with many unique computational steps can be transformed to alter the working-set granularity through step fusion or distribution. These transformations are similar to classic loop fusion and distribution, providing benefits through computation reordering while preserving execution semantics. Furthermore, we look at tiling the computation steps to increase the working set size and reduce the amount of fine-grained parallelism. These techniques help reduce scheduling overhead and improve data locality for stencil codes such as LULESH. In our experiments, we show that step fusion and tiling greatly impact performance and scalability in the mini-app, giving up to 38x speedup over the sequential baseline implementation on 48 cores for a moderate (60^3) simulation size.

2. OVERVIEW OF CNC

In this section, we provide a brief overview of Concurrent Collections (CnC). We summarize the basic concepts and highlight the important aspects that make CnC a compelling programming model for scientific applications such as LULESH. A more in-depth description of CnC can be

WOLFHPC2015 November 15-20 2015, Austin, TX, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM
978-1-4503-4016-8/15/11 ...$15.00.
http://dx.doi.org/10.1145/2830018.2830024..

found from previous related work [2, 3, 14].

The CnC programming model separates the algorithmic semantics of a program from how the program is executed. This is an attractive solution for a domain scientist, whose primary concern is focused on algorithmic correctness and stability. CnC relies on a tuning expert, whose expertise is in performance, to properly map the application to a specific platform. This separation of concerns simplifies the roles of each party. The domain scientist can program without needing to reason about parallelism or architecture constraints, while the tuning expert can focus on performance given the maximum freedom to map the execution onto the hardware.

The restrictions are that CnC programs must adhere to a certain set of rules. Firstly, computation units (steps) in CnC must execute statelessly, so they cannot modify any global data. Instead, the language uses built-in put/get operations to store and retrieve data items, which assist the runtime with scheduling dependencies. Secondly, all data items exhibit the dynamic single-assignment property, meaning they are immutable once they are written to. This requirement prevents data races during parallel execution and helps resolve data dependencies during runtime. In the following section, we give a brief overview of the CnC programming model.

CnC Programming Model

We describe the CnC language using a simple example of a reduction routine. Assume that we wish to sum values from four different sources, $F = \sum_{i=1}^{4} x_i$. Figure 1 shows a CnC representation of how a domain expert may represent the computation and its dependencies.

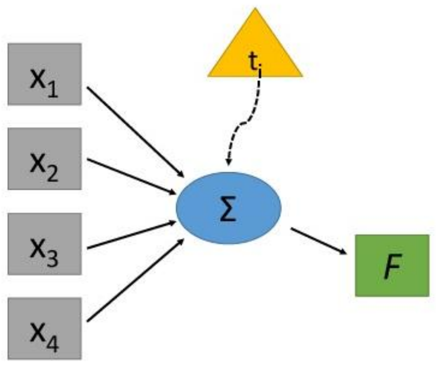

Figure 1: Reduction Operation in CnC

There are three types of nodes in this figure: computation steps, data items, and control tags. The solid directed edges between nodes specify producer-consumer data dependencies in the program, while the dotted edge corresponds to a control dependency. The computational units in CnC are steps, represented as ovals in Figure 1. Steps are organized in step collections, which contains distinct stateless operations that need to be scheduled and executed. In CnC, steps are dynamically created when control tags are *prescribed* in the program. The set of control tags are saved in the tag collection. In our example, a tag is created for every reduction step that is needed to execute and is represented as the yellow triangle. Finally, the program data is stored in the (data) item collection, shown in Figure 1 by the rectangular nodes. The item collection contains instances of data that steps use as input or outputs. Each data item is dynamic and is atomically assigned once in its lifetime (dynamic single-assignment) in order to provide the runtime with definitive information and prevent race conditions. These data items are also important during execution for scheduling; many steps consume data from previous steps as well as produce them, indicating program dependencies.

In a typical CnC program, the step collection will be defined similarly to a traditional C function, except it cannot hold any state and must receive data from other steps through data items. For our reduction, the program will first prescribe the reduction computation step using a control tag t_i. Once prescribed, the step will be scheduled to run once all data dependencies are resolved. In the example, the reduction step waits to *get* 4 values, x_1 to x_4, indicating the function will consume those values produced from a previous computation. Just as the *get* construct indicates a consumer dependence, the producers of the data will produce those values by initiating a *put*. Once the computation is performed in the reduction step, it will also store the result F by initiating a *put* to its respective data item using a tag for future retreival. The *get* and *put* constructs assists the runtime in scheduling steps and resolving dependencies.

A step collection can only be associated with a single tag collection/space; however, a single tag collection can prescribe multiple step collections. Steps are created when a control tag prescribes a particular instance of the step collection. While the existence of a tag prescribed for a specific step means that instance of a step will execute, it does not specify when that step executes. Execution of that step depends on the availability of data items (consumer dependencies) as well as the CnC scheduler which determines how tasks are handled. How and when tags are assigned is left up to the programmer. CnC allows for prescribing tags in advance, leaving the scheduler manage dormant tags, as well as dynamically prescribing tags for each step instance as their data becomes ready. Previous work shows tradeoffs exist for each implementation, but it is a flexible parameter that can be tuned [4].

Tasks are scheduled as the program executes, with steps producing data and sometimes prescribing tags. A step becomes available as soon as its tag is prescribed, and produced data items become ready once they are atomically written. Whenever item dependencies are ready and the tag has been prescribed, a step becomes enabled and may be executed. The runtime exploits parallelism by scheduling the available step instances as soon as they become enabled. Once all prescribed steps have been executed, the CnC program completes.

In principle, CnC programs should be well-tailored to run on both shared memory or distributed systems [5]. Furthermore, there is support for various different schedulers for the CnC runtime. We focus on using the Intel CnC Thread Building Blocks (TBB) work-stealing implementation, which uses Cilk style work stealing on top of the default work queues [6, 15]. Although alternative schedulers exist, we find that the TBB scheduler consistently outperforms or performs almost as well as the best schedule for most cases.

Our work uses the Intel CnC implementation, being one of the more mature and robust versions of CnC [5]. In this version, there is an additional speculation mechanism built into the runtime. As threads are assigned steps to execute, sometimes a step will begin execution before all inputs are ready. In this scenario, the runtime will requeue the step and tries it again at a later time. We discuss the effects of requeuing later in the results.

3. OVERVIEW OF LULESH

In this section, we describe the LULESH (2.0) application and the details of the algorithm written in CnC. LULESH is a fully-featured hydrodynamics mini-app developed by Lawrence Livermore National Laboratory that simulates the effect of a blast wave in a physical domain through explicit time-stepping. Hydrodynamics is a challenging problem, and was previously shown to account for a significant fraction (27%) of computing resources used by the Department of Defense [9]. We use LULESH to test optimization techniques that can likely be applied to problems in other domains that share the same static/dynamics properties with time-stepping execution.

The original LULESH specification is physics code that operates on an unstructured hexahedral mesh with two centerings. There is an element centering (center of the hexahedral) that stores data on thermodynamic and physical properties and a nodal centering (the corners of each hexahedra) that tracks kinetic values such as position and velocity of those points. These 2 centerings are also the primary iteration spaces involved in the program, and form the foundation for the CnC tag collection.

The application begins by initializing a 3-dimensional hexahedral mesh of arbitrary size, then defining the spatial coordinates and neighbors of all elements and nodes in the mesh. Surrounding boundary conditions are applied and initial values are assigned before beginning the time-stepping algorithm. At every iteration, the time scale is computed based on physical constraints to determine a maximum safe time value to advance for the subsequent iteration. Once determined, the kinetic values (force, position, velocity) are computed for all nodal quantities, which in turn are used to calculate the thermodynamic variables for all elements. The cycle completes as the next time-step value is calculated using the constraint values from the current step. More detailed information about the LULESH mini-app can be found in other papers [9, 11, 12].

3.1 CnC Specification

Every CnC program includes a high level skeleton called the CnC specification, which can also be referred to as a *domain specification*. This specification contains information regarding the item, step, and tag collections, as well as all data produced and consumed by each step. Using this data, a high level dependency graph can be constructed to analyze most of the data and control flow. Initially, we believed this information was enough to make high level program changes by applying graph-based transformations. However, it became apparent that the data available was insufficient to apply high level transformations due to lower-level program interactions.

The high level algorithm of LULESH can be seen Figure 2, represented in a decomposed form consisting of algorithmic steps. Each node in the graph represents an important computational step involved in the LULESH algorithm, with directed edges correspond to producer/consumer data dependencies for each step. For simplicity, only dependencies for current time steps are shown, since dependencies from previous time steps are trivial and do not deter the algorithm's control flow. We list and give a brief summary of each computational step.

- Compute Delta Time: Prior to every iteration, this checks all element data from the previous iteration to determine the next time step value. Has a separate tag space.
- Compute Stress/Hourglass Partial Force: Forces are calculated for each element using data from the previous iteration's elements.
- Force Reduction: Partial forces for every node are summed up from 8 neighboring elements.
- Compute Velocity/Position: Kinetic values are computed for each node using previous nodal forces/positions/velocities.
- Compute Volume/Derivative/Gradient/Characteristic: Physical properties are computed for each element using kinetic values.
- Compute Viscosity Terms: Neighbor gradient data is gathered along with volume data to calculate element viscosity terms.
- Compute Energy Terms/Time Constraints: Thermodynamics/Physics terms are calculated for each element using previous element data.

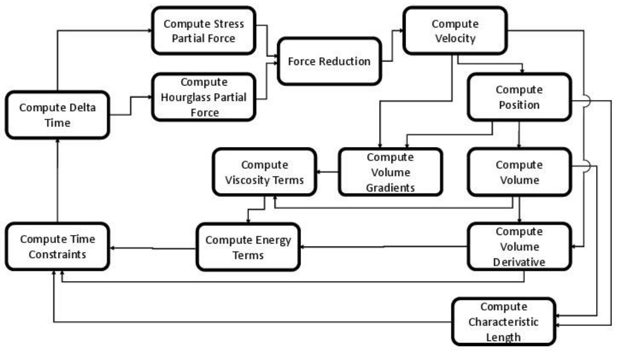

Figure 2: High-level LULESH Algorithm

This information is also depicted in CnC's textual representation of the *CnC specification* for the program, as shown in Figure 3, which contains the high level information of each collection, including the list of steps, tag declarations, and data items produced and consumed by each step. In CnC programs, the user must explicitly declare all items for each 3 collections. This information is helpful for determining the available optimizations that can be performed to transform the code, but not sufficient to determine the legality of all transformations. In the following sections, we describe the different transformations that can be legally made in CnC to optimize its performance.

The two techniques we focus on are step fusion and step tiling. Conceptually, the ideas are similar to that of loop fusion and tiling in classical compilers, except in the context of CnC. One of the drawbacks of the CnC runtime is its inability to efficiently handle fine-grained parallelism due to its dynamic scheduling framework. Both step fusion and

```
struct lulesh_context:public
    context<lulesh_context>{

// Step Collections
step_collection<compute_dt>
    step_compute_dt;
step_collection<reduce_force>
    step_reduce_force;
...

// Item Collections
// per node items
item_collection<pair,vector>force;
item_collection<pair,vertex>position;
item_collection<pair,vector>velocity;
// per element items
...

// Tag Collections
tag_collection<pair>iteration_node;
tag_collection<pair>iteration_element;
tag_collection<int>iteration;
...

// Producer Dependencies
step_compute_dt.consumes(dt);
...

// Consumer Dependencies
step_compute_dt.produces(dt);
...
```

Figure 3: LULESH CnC Specification

tiling help alleviate the effects of excessive synchronization and scheduling costs without modifying the underlying program semantics. Although CnC has built-in tuners, almost none of these tuners alter the organization of the algorithm itself; most tuners operate at a much lower level to benefit the CnC runtime with things like memory usage, garbage collection, serialization/synchronization, and thread management. The closest alternative is the tag-range tuner, but it cannot handle step fusion and other optimizations such as removing redundant *get* operations from shared dependencies. The effects of our fusion and tiling techniques complement the back-end tuners provided in CnC. In the following sections, we explore the legality of fusion and tiling and how each technique affects performance in LULESH application.

3.2 Step Fusion

Step fusion is an effective way to serialize multiple steps in a CnC program without altering the computation. The primary restriction on fusion is that steps remain "step-like"—conceptually, once a step receives all of its inputs, it must be able to run to completion.

Given multiple step collections, these steps can be legally fused if and only if both step collections are prescribed using identical tags and all dependencies between fused steps are computed in previous steps generated by the same tag. Step fusion is illegal for steps prescribed from different tags, or if the resulting fused step would become a coroutine—in other words, if the execution of the step would require interleaving with another step, violating the step-like property.

In certain cases, data dependencies may come from step instances generated by different tags, such as a reduction operation, where a current step needs recently computed data from multiple instance of a previous step. Fusion is not legal between these two steps because we only serialize the computation for step instances of a single tag, not all instances of a previous one. When multiple steps are fused, data dependencies that exist between fused steps can be expressed as temporary data in the new step computation. The set of *get* data which are consumed from from each step fused are unioned to form a new set of dependencies for the fused step. In the collection space, it reduces the number of items in the step collections, but the number of data dependencies will likely increase per step.

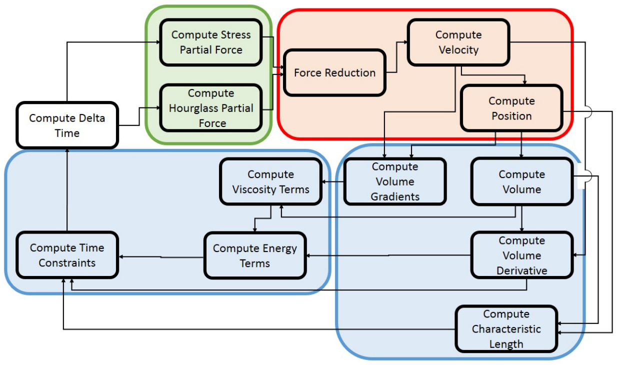

Figure 4: Fused LULESH Algorithm

We are able to apply step fusion in the CnC-LULESH program to reduce the number of step collection items from 13 to 5. Figure 4 highlights the steps in the algorithm where it is legal to fuse step collections. The leftmost node, *Compute Delta Time* requires its own space of tags per iteration due to the delta time calculation. The other steps are either in the nodal iteration space (red) or element iteration space (blue/green), each requiring separate tag collections. All the steps corresponding to the node tags, and most of the steps associated with the element tags can be fused. The two independent partial force calculations, shown in green, can be fused because they are independent and use the same element tags. They cannot be combined with the steps highlighted in blue even though they share the same tags because the force calculations depend on a delta-time calculation that is a global check that occurs at the beginning of each iteration and has a separate tag. The spatial computations that are prescribed by node tags can all be fused, reducing the three node steps into a single step, highlighted in red. For the rest of the element computation, fusion can reduce the remaining 6 element routines into 2 fused routines, as shown in blue in Figure 4. During the the viscosity step, there is a gather operation that requires data dependencies computed in a prior step using multiple different tags, implying a synchronization barrier and preventing those steps from legal fusion.

Initially, we believed using the consumer/producer dependencies expressed in the CnC specification would be sufficient to determine the legality of step fusion. However, that itself is not enough because of there can be cases where dependencies span across multiple instances of the same step prescribed from different tags, such as the element viscosity operation. Data items in CnC programs are written and read in an atomic fashion, thus naturally enforcing synchronization between dependencies. Step fusion alters execution semantics by changing what would be a synchronization step into serialization, but only for one tag instance. This is why steps which have dependencies coming from multiple tags cannot be fused.

3.3 Step tiling

We characterize step tiling as the coalescing of multiple tags to reduce the number of dynamic step instances. Tiling reduces the number of the tags, but increases the amount of work performed in each step. Similarly to fusion, step tiling serializes computation, altering the computation and the working set data. Instead of many dynamic step instances, a single step instance does the same amount of work, reducing scheduling overhead. This large step operates on a much larger working set, and in turn requires more memory. Additionally, step tiling alters the dependence structure. For larger tiles, the number of dependencies increases, and in some cases, neighboring data items may allow for data reuse, thereby reducing the number of *get* operations required. We find that step tiling is legal when there are no dependencies between different tags which prescribed that step. An example where tiling would be invalid for step collections would be Guass-Seidel, due to producer-consumer inter-dependencies between orthogonal tags for the same step.

There are almost no inter-tag dependencies in LULESH that prevent step tiling from being legal. Each set of tags correspond to independent nodes and elements of the mesh, and each step clearly defines the producer-consumer relationship between steps. Since CnC handles the scheduling aspect, the challenge is to correctly distribute and initialize the tags during the start of the program. There are also multiple ways to coalesce tags, and specifically for LULESH, we look at a blocking scheme and a strided per-row implementation. The block-tiling implementation partitions the problem into equal sized cubes that are a fraction of the size of the whole mesh, whereas the strided tiling implementation tiles the nodes and elements as long rows, where each row length as is equal to the mesh dimension. Figure 5 illustrates the difference between a block tile (red) and a strided tile (blue). When discussing tiling implementations, we will focus on these two tiling schemes.

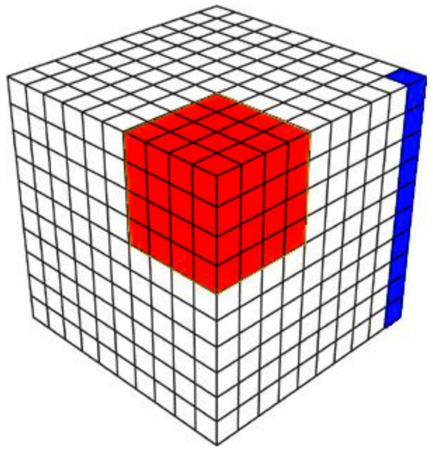

Figure 5: Blocked vs. Strided Tiling Example

The challenges of applying step fusion and tiling come from conforming to CnC coding standards. Blindly serializing multiple computations and iterations into a combined routine will likely modify the data and control flow within that new routine. However, good CnC performance relies on all CnC step collections being *step-like*, where all get operations occur at the beginning of the step (put operations can still occur freely). If *get* operations are do not occur first, the performance of the scheduler suffers, and in Intel-CnC, requeueing activity heavily impacts performance. Additionally, this coalescing increases the sizes of the working set which increases memory usage since there is more temporary data to save between computation.

Step tiling can be further tuned when dependencies are stencil-like such as in LULESH. In addition to spatial locality achieved from tiling nearby elements, there is significant re-use of dependencies due to nodal and elemental operations that require neighboring data. A single element operation would normally require 8 *get* operations. A naively tiled code could perform that 8 times, requiring 64 total *gets*. However, an optimized tiling scheme for the same 8 steps would only require 27 *get* operations. We explore this optimization in LULESH, but the effects are marginal after the extra calculation to figure out which data items are shared, compared to executing redundant *get* operations. For a shared memory machine, this is the case, but on systems where communication costs far outweigh local resources, removing extra data movement will be beneficial.

4. RESULTS

In this section, we evaluate the performance of the LULESH application in CnC, with various configurations of the baseline, fused, tiled, and two fused+tiled (blocked and strided) schemes. We compare their execution running on our shared-memory system running on up to 48 processors. The following implementations are tested:

Baseline - This version, originally created from researchers at the Pacific Northwest National Laboratory, describes the LULESH application at its most decomposed level, with the minimal constraints. Domain scientists would most likely want to express their algorithms this way. This follows CnCs principles of writing the program without expressing how it is parallelized. The burden falls to the CnC runtime, which is given the maximum amount of flexibility to optimize parallel execution. In this version, all tags are prescribed in advance and every stencil element performance every computation, requiring dynamic instances for each. Unfortunately, CnC is unable to effectively schedule stencil programs with that much fine-grain-parallelism, causing reduced performance without further tuning.

Fusion-only - This implementation fuses all computation that can be coalesced in each tag space. Step fusion is applied to the tags corresponding to the nodal and element iteration spaces, reducing the step collection size from 13 to 5. This step size reduction provides around a constant 2x speedup over the baseline implementation.

Tiling-only - This version of LULESH tiles the baseline implementation and coarsens the tag space, thereby reducing the number of created dynamic step instances. Every dynamic instance computes over a larger working data set, but also requires more intermediate memory. The size the working set is a tuneable parameter, which we investigate. Tiling improves overall program scalability and performance by reducing scheduling overhead and improving data locality. Neighboring data elements and nodes are grouped into

Table 1: Timing Results: 60^3 Sized Mesh

	Number of Cores						
	1	**2**	**4**	**8**	**12**	**32**	**48**
Baseline	53.180	52.900	71.852	108.531	110.515	110.572	119.466
Fusion Only	26.288	26.405	33.3389	52.224	51.391	54.340	57.430
Tiling Only	12.67596	12.5693	6.677482	4.234	2.652	2.504	2.526
Blocked Fuse-Tile	10.749	10.699	5.110	2.883	1.845	1.557	1.768
Strided Fuse-Tile	11.171	10.610	6.025	3.498	2.104	1.483	1.397

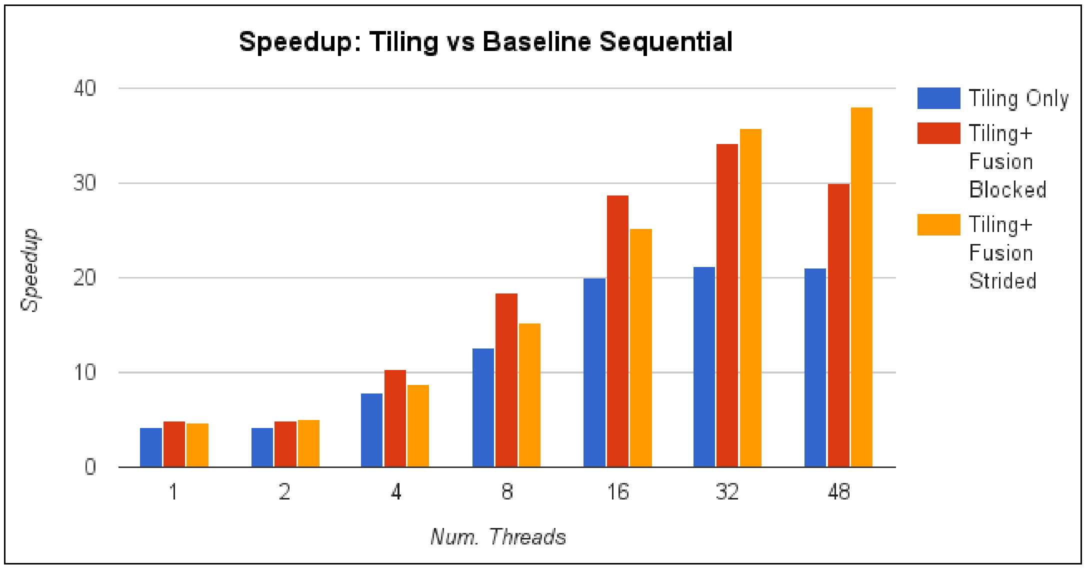

Figure 6: Scalability for Tiling Implementations

the same tile, allowing for data reuse computations that access shared data. The items in the data collections are minimally altered to maintain the overall structure of the original program.

Fused-Tiled - This implementation combines both the fusion and tiling optimizations. The step collections and tag collections are fused and tiled to create new larger steps. We also try to exploit the locality for shared data that exists for tags with shared common neighbors. The transformation requires some coding changes which require bookkeeping to account for extra variables and computation re-ordering to preserve step-like properties required by every CnC step, where *gets* all occur at the beginning of a computation step.

We also note that there are specific tuners available in the Intel CnC implementation that would have eased programmability for fusion and tiling implementations. One of these includes the *depends* tuner. This tuner prevents preemptive scheduling of a step collection until all variables are ready.

Evaluation

Experiments were run on mesh sizes up to size 60 for 30 iterations, ten times per heuristic, with minimum and maximum results excluded to reduce variance. The hardware is a shared memory, AMD Opteron 6176 SE system configured with four 12-core processors (48 cores total) running at 2.3 GHz, with 512 KB per-core level 2 cache, and 12 MB level 3 cache. Table 1 shows the timing results per-iteration for a mesh of dimension 60^3. In the LULESH manual, the authors recommend experiments to run for problem sizes around 50-90 along each dimension, and experiments with a problem size of 60 provides us with consistent results in that range while allowing for even sized block partitions [12].

The baseline implementation performs poorly in CnC. With the baseline program, CnC creates 60^3 dynamic step instances for each minimally-constrained step. Such fine granularity requires CnC to perform excessive scheduling and bookkeeping during execution, even causing it to perform worse than in sequential. Applying step fusion halves the number of steps, and results in a 2x speedup. Fusion by itself does not impact scalability until tiling is consider. A much larger benefit can be seen from step tiling, where we see speedups over 20x faster than sequential when running on 48 threads, for the tiling-only implementation. Figure 6 shows the scaling for the three tiling implementations. Step tiling gives a significant speedup compared to fusion, scaling with higher thread counts, but does not scale well past 16 threads. However, combining both step fusion and tiling gives the greatest performance increase, giving up to 37x speedup over the sequential version. It is interesting to see the block tiling method perform better than the strided implementation at lower thread counts. We reason that this is due to larger tile sizes in the block tiles, which are 15 in each dimension. Larger tiles coarsens the partitioning, reducing the available parallelism, but performs better for a smaller thread count. However, at 32 and 48 threads, strided tiling, which offers more parallelism and better tile parity, scales better than the blocking method.

We further investigate the impact of step tiling and two different tiling implementations in LULESH. The first method, strided tiling, groups full rows of nodes or elements along one dimension of the mesh into a single block of computation. These long strides scale to all mesh sizes and the transition between node-based and element-based computations is seamless. Every element tile depends on exactly four neighboring node tiles when executing each iteration. Strided tiling is able to create equivalent sized tiles, although at the cost of being finer grain and no way to adjust the granularity.

The block tiling method partitions the element space of the original mesh into equivalent sized subdomains. However, the node space cannot be mapped directly onto the element partitions directly, and are partitioned similarly to strided tiling. In an ideal scenario, all nodal dependencies would fit into a single tile which would map perfectly onto each element tile, so that all dependencies would come from a single tile. However, because of shared neighbors and the offset sizes between nodes and elements, it is extremely difficult to map properly under the CnC framework, which expects dynamic single assignment of data. Alternative solutions add increased complexity to the partitioning setup and/or more redundant computation, so we do not include those results. Even with an imperfect mapping, there is ample parallelism in the application, but we believe that the scalability suffers due to it. The parity between node and element tiles in the strided method shows to give benefits in terms of scalability. After 16 cores, block tiling becomes inferior to strided tiling, most likely due to this reason.

While strided row-tiles are exactly the size of the mesh dimension, block tiles do not have restrictions on their sizes, provided that the data and corresponding tags are initialized correctly. We investigate 4 different sizes for a mesh of size 60^3, with results shown in Figure 7. When executing using less than 8 threads, the behavior is similar to the performance running with 8 threads, with the larger block sizes being superior. However, we can see that after using more than 8 threads, it becomes more advantageous to use smaller block sizes, creating more fine-grained parallel units. Depending on the problem size, this is most likely a tuneable parameter.

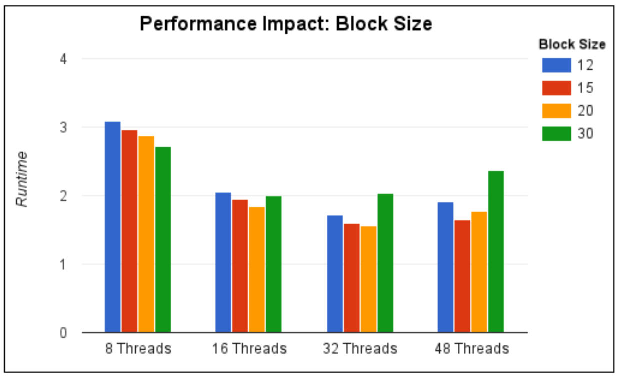

Figure 7: Impact of Block Size

5. DISCUSSION

The final fused-tiled implementation had a number of improvements over the baseline code. As stated previously, the number of get and put operations were reduced, as well as the number of tags that were prescribed. Additionally, some data calculations were able to be reused, and a few dependencies were serialized which reduced scheduling synchronization. During fusion and tiling modifications, we reordered specific read/write operations to best accommodate CnC's functional guidelines, which gave a slight performance boost and reduced requeue events. It should be noted that two premature implementations were tested which did not perform as well as the final heuristics. The first method thoughtlessly fused and/or tiled the step functions and relied on a built-in tuner to handle the dependencies, allowing non-ideal orderings of get and put operations in each function. This *Depends* tuner prevents speculation, which negates any negative requeuing effect. Although it provided a simple programming solution, the tuner was too restrictive and the resulting program never scaled when running in parallel. The second iteration included the reordering of get and put operations, but did not manage memory and temporary values effectively. With some additional garbage collection and the removal of redundancy between shared neighboring data, we were able to get a 10% speedup over a non-optimized code.

Although the CnC port is based on the original LULESH 2.0 implementations provided by LLNL, the performance, especially sequential code, does not perform as well. This was one of the original motivations for the work, but the transformations discussed can be generalized to other applications as well. When comparing, we found that the CnC sequential baseline is far worse than its OpenMP alternative. With optimizations, our CnC implementation is around 10x slower than the later versions of the LLNL code. Upon further investigation, we reason the degredation is primarily caused by cache performance from the lack of cache-specific tuners for our tiled/fused implementation. Additionally, there are greater overheads with accessing individual elements from the data collection during CnC's dynamic execution of steps, compared to OpenMP which operates on parallel regions with uniform memory access and does not dynamically manage dependencies. With underlying changes to the item collections and a modified data layout to match the computation tiles, we believe removing the bottlenecks associated with data access and movement will give us performance much closer to that of the LULESH 2.0 code.

An execution trace is shown during parallel execution of 8 threads, shown in Figure 8. Each block represents an active task working on a piece of computation, for the fused-tiled LULESH program, but on a smaller 20^3 problem. This demonstrates that LULESH is a complex application to handle in CnC, especially in parallel. The gaps between the partial force calculations are mostly due to requeued events, which occur in Intel CnC when dependencies are not ready for a specific step collection. The data dependencies before and after the force calculations require a reduction operation, effectively becoming a synchronization barrier. As seen by the trace, the element-based computations afterwards happen without much idle time in between. Finally, we note that in between iterations, the delta time computation requires a significant fraction per iteration. This step is a

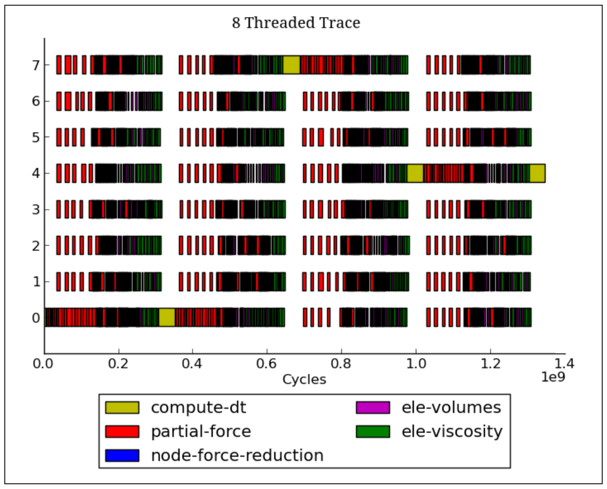

Figure 8: Multi-threaded CnC Execution Trace

safety check which requires every element to have calculated all values for the previous iteration before advancing to the next one. With CnC, which exploits asynchronous-parallel execution, we achieve between a 70-80% utilization ratio but are unable to fully take advantage of maximum parallelism due to implicit barriers caused by the dependence structure in LULESH.

5.1 Lessons Learned

In our study, we focus on a single application, LULESH. From the minimally constrained algorithm written in CnC, we applied high level fusion and tiling transformations on the program by altering the step, data, and tag collections while making negligible changes to the underlying data structures and low level code. However, the applicability of this step fusion and tiling technique are not limited to the single application. These techniques should be applicable to any high level algorithm using the CnC language and are not domain-specific. The advantages of using the CnC programming model creates a separation of concerns that allows domain scientists to focus on the algorithm, while the tuning expert can focus on performance concerns such as tiling and fusion. However, the current CnC tuning capabilities do not always yield scalable applications using the CnC language if the algorithm specifies a level of parallelism that is too fine. We show this using the LULESH mini-app and exploit the fact that there was enough algorithmic complexity to take advantage of step fusion. Being a 3D stencil-based code, it also benefits from tiling techniques to exploit the available parallelism in each iteration. When it comes to partitioning the iteration space of CnC steps, fusion and tiling are semantically equivalent to combining multiple steps into a single instance. It is a very general approach, and the transformations do not alter any computations, but only change how and when they are executed.

Although the information from CnCs high level specification can aid in the transformations, the dependency information is insufficient to fully automate the process. Parsing the CnC graph does not provide enough data to analyze the dependence structure between individual step instances. In the future, we wish to automate fusion and tiling techniques using the information from the high level CnC specification as well as inter-tag dependencies based on *get* and *put* indices. We believe that this is possible, but more information must be available at the specification level in order to detect when dependencies cross between tag instances. Preliminary results using CnC-OCR, a separate CnC implementation, has shown promise in generating a bare-bones CnC program which automatically interfaces every step collection to its respective data collections [8]. This feature requires dependencies to be explicitly stated, but variable tile sizes will require underlying data structure changes. Future work in that direction will hopefully produce a process to automate step tiling that will allow fusion and other optimizations.

6. RELATED WORK

Parallel Programming is still a difficult task, even after so many decades. Legacy codes are difficult to maintain; meanwhile, no one is certain what our hardware will look like in 10 years. Many researchers are still trying to figure out the best trade-off between programming portability and performance. Concurrent Collections is just one approach for efficiently program parallel applications.

6.1 Scientific Applications

One well-known approach to programming scientific applications is through the use of domain-specific languages (DSLs). Previously, Karlin et al. [10] had explored the applicability of using traditional and emerging parallel programming models for LULESH. These included Chapel, `Charm++`, Liszt, and Loci. Their results were mixed, citing that no single model handled all optimizations perfectly, and that only the longstanding frameworks performed competitively due to having more time to mature runtime and compiler technologies. However, the newer technologies mostly utilized high-level programming constructs and provided easier programmability and better tuning interfaces, with better potential for programmers to obtain reasonable performance using significantly less time and effort.

CnC would fall into the category of offering superior ease of programmability and portability for reasonable performance. Previous work evaluating the performance of CnC on a Cholesky Factorization yielded very competitive results compared to highly tuned MKL kernels [4]. In their study, they acknowledge that performance is impacted by tile-size and locality, and observe minor performance degradation from queuing and scheduling overheads. We surmise that with enough tuning to taking advantage of Intel's optimizations, we could achieve competitive performance rivaling tuned versions of LULESH.

6.2 Parallel Programming Models

CnC goes beyond just that of scientific applications. The core belief in CnC is that any parallel application can be programmed using their model, which was designed specifically for that cause. The semantics of parallelism is expressed without explicitly determining any parallel execution, simplifying the programming process for non-performance experts. The issue that arises is how to optimally express the computation granularity so that it an be tuned efficiently. Our approach focuses on tiling/fusion techniques and high-level tuning in CnC to coarsen the iteration space and balance the work/scheduling ratio to aid the runtime.

Another alternative to exploiting parallelism is through the use of polyhedral frameworks. Polyhedral frameworks are very powerful compiler tools for analyzing and transforming loop-based codes. They can reason about separate iterations in loops and analyze symbolic expressions, unlike more traditional compiler techniques. PLUTO, an automatic parallelizer based on the polyhedral model, focuses specifically on addressing tiling and locality concerns [1]. We note that the notion of tiling they use is the general term addressed for common loops, and not the tiling we focus on, in the CnC space. Another similar work that has connections to both CnC and polyhedral compilation frameworks is Data Flow Graph Representation (DFGR), an intermediate graph representation for macro-dataflow programs [16]. In their work, Sbirlea et al. uses program information similar to the CnC specification to help her framework generate scalable parallel programs that run using various libraries. One of their purposes for relying on the framework is to generate well-tiled code, but do not need to tune any tiling parameters.

CnC is still a relatively new programming model. There are two major implementations, with Intel's version and another version being developed by Rice that runs using their Open Community Runtime (OCR) [8]. Future work will include trying to automate the process of fusion and tiling for programs using either CnC framework. Another idea being explored is to have a dynamic runtime that can leverage the benefits of fusion and tiling on demand to run the original, fused, tiled, or both, depending on which implementation will be more advantageous. This increases platform flexibility, potential dynamic adaptation, but will also likely add increased complexity along with unknown feasibility.

7. CONCLUSION

In this paper, we present a transformative method in Concurrent Collections to optimize the performance of the LULESH mini-app, a hydrodynamics stencil code developed by LLNL. Borrowing from classical compiler concepts, we apply fusion and tiling onto the fundamentals core components of CnC programs, altering only the collections while making negligible changes to underlying code. This produced a semantically equivalent algorithm while reducing the granularity of the program. Although Concurrent Collections offers intuitive programming simplifications, it has trouble with over-scheduling programs (like stencils) with fine-grained parallelism. We start with what a scientist understands, beginning with a decomposed algorithm consisting of minimally constrained computational steps, and are able to transform that program into a well-optimized CnC program that is scalable and architecture-agnostic.

To achieve good performance in CnC, the programmer has to more than just express the partially ordered set of computations. If a CnC program has extremely fine-grained parallelism, no amount of built-in performance tuning will prevent severe performance degradation. For most decomposed algorithms for stencil codes similar to LULESH, step fusion and step tiling or an equivalent programming change will be necessary to obtain scalable performance on modern hardware.

Acknowledgments.

This research is supported by the Department of Energy under contract DE-FC02-12ER26104. We would like to thank the anonymous reviewers for their feedback and suggestions. We would also like to thank Ellen Porter and her work at the Pacific Northwest National Laboratory for providing us with the original LULESH program, as well as Kath Knobe, Nick Vrvilo, Frank Schlimbach, and Zoran Budimlic, for their comments and feedback during discussions regarding Concurrent Collections.

References

[1] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. Pluto: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer, 2008.

[2] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, et al. Concurrent collections. *Scientific Programming*, 18(3-4):203–217, 2010.

[3] M. G. Burke, K. Knobe, R. Newton, and V. Sarkar. Concurrent collections programming model. In *Encyclopedia of Parallel Computing*, pages 364–371. Springer, 2011.

[4] A. Chandramowlishwaran, K. Knobe, and R. Vuduc. Performance evaluation of concurrent collections on high-performance multicore computing systems. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[5] I. C. Frank Schlimbach. Intel concurrent collections for c++ for windows and linux, 2015.

[6] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *ACM Sigplan Notices*, volume 33, pages 212–223. ACM, 1998.

[7] D. Gelernter. *Multiple tuple spaces in Linda*. Springer, 1989.

[8] Habanero-Rice. Concurrent collections on ocr, 2015.

[9] I. Karlin, A. Bhatele, B. L. Chamberlain, J. Cohen, Z. Devito, M. Gokhale, R. Haque, R. Hornung, J. Keasler, D. Laney, et al. Lulesh programming model and performance ports overview. *Lawrence Livermore National Laboratory (LLNL), Livermore, CA, Tech. Rep*, 2012.

[10] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, et al. Exploring traditional and emerging parallel programming models using a proxy application. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 919–932. IEEE, 2013.

[11] I. Karlin, J. Keasler, and R. Neely. Lulesh 2.0 updates and changes. *Livermore, CAAugust*, 2013.

[12] I. Karlin, J. McGraw, J. Keasler, and B. Still. Tuning the lulesh mini-app for current and future hardware. In *Nuclear Explosive Code Development Conference Proceedings (NECDC12)*, 2012.

[13] K. Knobe. Ease of use with concurrent collections

(cnc). *Hot Topics in Parallelism*, 2009.

[14] K. Knobe and Z. Budimlic. Compiler optimization of an application-specific runtime.

[15] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.", 2007.

[16] A. Sbirlea, L.-N. Pouchet, and V. Sarkar. Dfgr an intermediate graph representation for macro-dataflow programs. In *Data-Flow Execution Models for Extreme Scale Computing (DFM), 2014 Fourth Workshop on*, pages 38–45. IEEE, 2014.