Evaluating Performance of Task and Data Coarsening in Concurrent Collections

Chenyang Liu and Milind Kulkarni

Purdue University, West Lafayette IN 47907, USA,
Liu441@purdue.edu, Milind@purdue.edu

Abstract. Programmers are faced with many challenges for obtaining performance on machines with increasingly capable, yet increasingly complex hardware. A trend towards task-parallel and asynchronous many-task programming models aim to alleviate the burden of parallel programming on a vast array of current and future platforms. One such model, Concurrent Collections (CnC), provides a programming paradigm that emphasizes the separation of the concerns–domain experts concentrate on their algorithms and correctness, whereas performance experts handle mapping and tuning to a target platform. Deep understanding of parallel constructs and behavior is not necessary to write parallel applications that will run on various multi-threaded and multi-core platforms when using the CnC model. However, performance can vary greatly depending on the the granularity of tasks and data declared by the programmer. These program-specific decisions are not part of the CnC tuning capabilities and must be tuned in the program. We analyze the performance behavior based on tuning various elements in each collection for the LULESH application using CnC. We demonstrate the effects of different techniques to modify task and data granularity in CnC collections. Our fully tiled CnC implementation outperforms the OpenMP counterpart by 3x for 48 processors. Finally, we propose guidelines to emulate the techniques used to obtain high performance while improving programmability.

Keywords: Concurrent Collections, LULESH, Coarsening, Parallel Programming

1 Introduction

Developing scientific applications for high performance computing is no easy task. Knowledge of the scientific domain is necessary in order to understand the underlying methods and equations required to solve the problem. Correctly mapping and distributing that algorithm onto modern parallel architectures is another task in itself. Modern clusters are increasingly sophisticated, with various forms of heterogeneous and homogeneous parallelism while sporting complex memory hierarchies. A recent emergence of high-level programming models aim to alleviate the burden of parallel programming on a vast array of future platforms. These frameworks, based on the asynchronous many-task model, split programs into smaller units of computation and associated dependencies, relying

on runtime schedulers to correctly synchronize task execution. The programming model we explore is Concurrent Collections (CnC), which is a data-driven task-parallel programming model designed to change the way we approach parallel programming.

The key motivation for developing in CnC is its *separation of concerns* philosophy. The concerns of the domain expert, whose knowledge is used to correctly develop the method and algorithm, is separated from that of the performance expert, whose strengths are in hardware and software optimization. Programs using CnC are expressed as a partially-ordered set of computations with explicitly defined dependencies and seamlessly exploit parallelism by following the constraints of data dependencies using a data driven approach. The CnC scheduler synchronizes data and maps computational tasks to the target hardware at runtime. However, dynamic runtime mapping does not always yield high performance for the following reasons. Excessive fine-grain-parallelism will overburden the scheduler, while sub-optimal data movement leads to poor memory performance. In this paper, we analyze and quantify the effects of high level changes to the granularity of collection items in CnC programs. We use step fusion and tag tiling to coarsening task parallelism, while data is tiled to match the larger computation blocks. We perform these optimizations on the Livermore Unstructured Lagrange Explicit Shock Hydro (LULESH) mini-app, a hydrodynamics code created in the DARPA UHPC program [1, 2].

We present the LULESH application, starting from a minimally constrained implementation, and analyze opportunities to reduce the fine-grained parallelism through step fusion and tag tiling. Previous work has shown that these high level techniques improves performance, but does not outperform simple interfaces such as OpenMP [3]. However, with homogeneous tiling of the coarsened execution along with data items, our optimized LULESH implementation outperforms the LULESH 2.0.3 with OpenMP directives by 3x on 48 cores for a 60^3 sized problem. Finally, we present a recommended method for writing CnC programs using automation tools for setting up CnC directives for increased programming productivity.

2 Concurrent Collections Model

In this section, we provide some background on the Concurrent Collections (CnC) programming model. We discuss the methodology for writing programs using CnC and explain how it achieves its philosophy of separation of concerns, making it a compelling model to use for programming applications such as LULESH. A more in-depth description of CnC can be found in previous works [4, 5].

Unlike traditional programming approaches, the CnC programming paradigm avoids expressing control flow or parallelism in its program structure. CnC replaces the need for threads and locks or parallel regions, instead satisfying dependence constraints using a data-driven execution model to exploit parallelism. This model is an attractive solution for a domain scientists, whose concern is focused on algorithmic correctness and stability. In contrast, CnC employs various

tuners for performance experts to best map certain aspects of an application to target platforms. These tuners are often used for machine-specific optimizations such as memory locality, thread affinity, and resource mapping for distributed applications [6]. CnC is also compatible with a number of programming languages including C/C++, Python, Scala, and Haskell, and also supports various back-end runtime frameworks such as Intel's Thread Building Blocks (TBB) library, Open Community Runtime (OCR), and CnC-HC for GPUs [7, 8, 9]. In our research, we use the C++ interface along with the Intel runtime and TBB based work-stealing scheduler for its robustness and tendency to outperform the other schedulers.

There are three basic building blocks that constitute a CnC program. These are referred to as the *collections*, whose purposes are to establish the computation *steps* being performed, *tag* and prescribe those step with unique identifiers, and express which *data* are consumed and produced by computation steps. Figure 1 depicts the three collections and their relationships along with a high level overview of the data-driven execution in CnC.

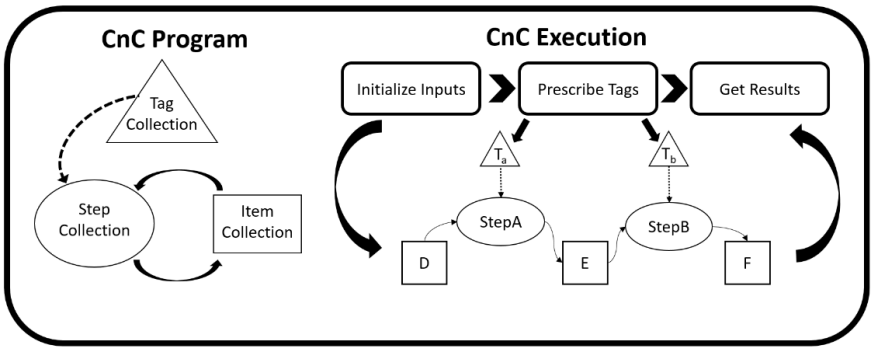

Fig. 1. CnC Program and Execution

The three types of nodes in Figure 1 correspond to the computation steps (ovals), data items (rectangles), and control tags (triangles) in CnC. The step collection contains stateless computation steps of a program which are dynamically instantiated when control tags such as T_a *prescribe* those steps. This collection of tags usually contains temporal/spacial data to assist with control flow and proper execution for dynamic steps. Finally, the set of producer/consumer dependencies comprises the (data) item collection. Data items follow dynamic single-assignment, meaning they are immutable, but elements in the data (item) collection may have multiple dynamic instantiations using unique handles, similar to hashing key/value pairs.

The step collection contains a program's computational steps, similar to that in traditional functions. However, these steps do not modify global data, and input/output dependencies are handled by CnC constructs. Steps routines must use *get* constructs to access/consume data (item collection) inputs and *put* constructs to write/produce updated values. Valid steps must perform all *get* oper-

ations at the start of each step and *put* operations may only occur after all *gets* finish. Additionally, each step may only have a single associated tag, but a single tag may prescribe multiple steps. Steps will execute when a tag has prescribed it, and all data dependencies are ready from previous steps or the environment.

While step collections specify the computation on data, control tags dictate which steps are dynamically created during runtime. Tags can prescribe steps at any time in the program, whether it be dynamically during runtime or during program initialization. However, once a step is prescribed, the CnC runtime will ensure that step executes before program completion. In Figure 1, *StepA* begins execution only once tag T_a prescribes it and D is supplied by the environment. Similarly, *StepB* will not begin executing until tag T_b prescribes it and *StepA* finishes producing the data for E. The CnC program terminates once the last prescribed step is finished executing.

Conceptually, the CnC model is ideal for programmability on parallel platforms; however, shifting too much burden from the programmers to the runtime may become prohibitive for performance. After investigation, we find that expressing algorithms as steps that correspond to equations of a method does not translate into an efficient CnC program, unless task and data granularity are considered. Tuners are not sufficient because they mainly focus on machine-specific optimizations, whereas opportunities to reduce the runtime overheads rely on coarsening the task and data granularity, which depend on program structure.

3 LULESH Overview

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 by explicit time-stepping [1]. LULESH is a complex algorithm which performs both computation and communication based work, and optimizations in its code should apply similarly to other applications which exhibit stencil-like and/or time-stepping behavior.

The LULESH 2.0 specification is physics code that operates on an unstructured hexahedral mesh with two centerings. The element centerings (center of the hexahedral) handles data for thermodynamic and physical properties whereas the nodal centerings (the corners of each hexahedra) track spatial and kinetic values such as the position and velocity. The application begins by initializing a 3-dimensional hexahedral mesh and initializing components for each centering. The time-stepping begins as a force is then applied at the origin, updating the kinetic values for all the nodal centerings. Once nodal computation completes, a series of element-centered computations occurs, updating the thermodynamic variables for all elements. More in-depth papers describing the LULESH algorithm can be found in previous work by Karlin et al. [1, 2].

One key observation is that a great deal of computation is performed each iteration for both centerings. Furthermore, several computations are 3-dimensional stencil calculations that require neighboring communication, which due to the dual-centered scheme, creates unique challenges for optimization. Additionally,

there are producer/consumer relationships that span across cycles of time-stepping, making data synchronization a likely bottleneck. These unique characteristics present more opportunities for optimization unlike those in traditional *(AxPy)* matrix computation.

3.1 The LULESH Domain Specification

Following the CnC philosophy of separation of concerns, we map the LULESH algorithm as a high level graph, with computation steps and producer/consumer dependencies labeled. This *domain specification* of LULESH represents how a domain scientist might describe the algorithm, as seen in Figure 2. Each node in the graph represents a vital computational required by the algorithm, and the edges clearly depict from which steps that data is being produced and consumed for. We list and give a brief description of each computational step.

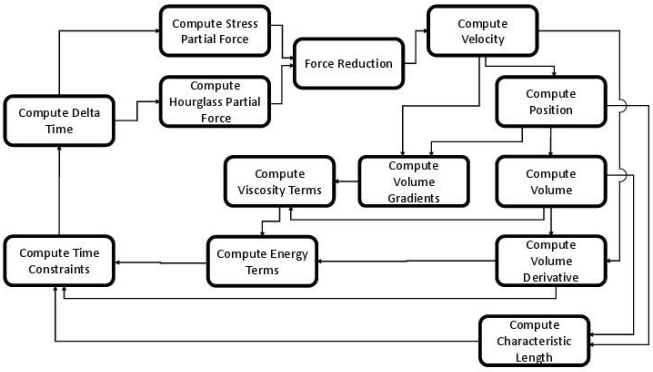

Fig. 2. High-level LULESH Algorithm

- **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**: Previous values and gradient data from 6 element neighbors is used to calculate element viscosity terms.
- **Compute Energy Terms/Time Constraints**: Thermodynamics/Physics terms are calculated for each element using previous element data.

Using the *domain specification*, a direct translation is made to the *CnC specification*, which is a textual representation describing the step, tag, and data collections. The *CnC specification* defines and declares most of the high level information inside the CnC *context* required in the program. Whereas step computation and data are the norm in traditional programming, tags are conceptually different. In the context, tags are declared along with which steps they prescribe. The number of prescribed steps and unique tag identifiers are not required for declaration; this occurs during runtime. In the following sections, we discuss our approach for optimizing this minimally constrained LULESH implementation.

Our baseline uses a CnC specification identical to that in Figure 2. Three sets of tags are used: per iteration, per node centering, and per element centering. Every step computation performs its required computation according to the hydrodynamics method, but the concerns for task granularity are neglected. In the following sections, we describe the coarsening techniques for each collection and its performance impact, with the task coarsening based on previous work [10]. However, that work was incomplete due to the lack of cohesive tiling with the data item collection members, which we include.

3.2 Step Fusion

Step fusion is an effective way to serialize multiple steps in a CnC program without altering the underlying computation. The decomposed LULESH algorithm has steps that operate on node and element centerings. Steps that share the same tag and operate on the same data can be legally fused, creating a new legal algorithm, as seen in Figure 3. However, this fusion is only legal when dependencies from previous steps are guaranteed to be ready under serial execution, or if the resulting fused step would require interleaving with another step (or itself) and become a coroutine. Therefore, computation requiring updated neighbor data such as ghost exchanges cannot be fused because the data will likely come from a step prescribed from a separate tag. When steps are fused, data dependencies that exist between original steps are serialized in the fused step. The set of producer/consumer data dependencies from each step are joined and become the new set of producer/consumer dependencies for the fused step.

Step fusion is applied to the CnC-LULESH program to reduce the number of step collection items from 13 down to 5. Figure 3 highlights the step computations that get fused in the updated algorithm. The leftmost node, *Compute Delta Time* requires its own space of tags per iteration due to the delta time calculation, but the other steps are either in the nodal iteration space (red) or element iteration space (blue/green), and can be properly fused. We fuse the force computations (green) which require element-wise computations for all elements, as well as the spatial/kinetic steps which operate on nodes (red). Fusing the force computation reduces parallelism, but it is helpful in our case where abundant parallelism exists. Also, the bottom 6 element computations (blue) can only be fused into 2 routines, due to ghost exchanges at the viscosity step, requiring data dependencies computed from the prior gradient step from multiple neighboring elements, thus preventing legal fusion.

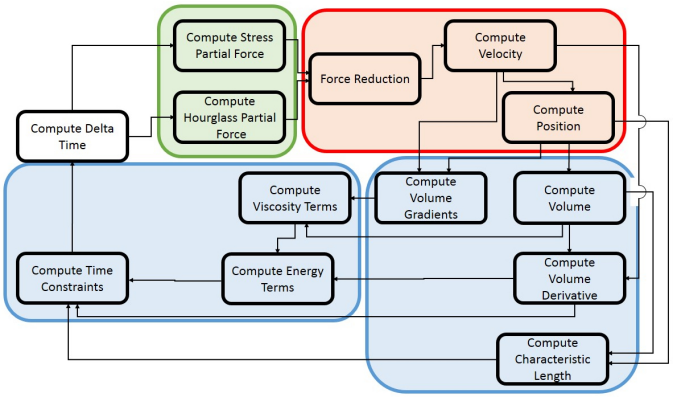

Fig. 3. Fused LULESH Algorithm

3.3 Tag Tiling

Tag tiling is an optimization used to reduce number of step prescriptions during execution. While a naive implementation would prescribe every step in the domain specification for each node and element, such a brute-force distribution would not scale to larger problem sizes. Tag tiling replaces multiple dynamic step instances by coalescing those tags into fewer larger step computations that span multiple tags. Similarly to step fusion, tag tiling serializes the computation in the new step. The new tiled computation will likely require large temporary working sets, as well as code modifications to reorder computation and optimize for potential locality.

In the LULESH code, we successfully tile all steps corresponding to the nodal and element-wise tags. Each tile contains a 3-dimensional spatial region that consists of the nodes or elements. Other tile shapes were considered, but we use hexahedral blocks to minimize the number of ghost regions when performing stencil updates. Implementing tag tiling involved minor changes to the steps themselves, as loops were introduced to handle additional work, step prescriptions were reduced, and indices remapped for correctness.

The effects of step fusion and tag tiling extend beyond just coarsening the task parallelism of the CnC program. The modified collections result in different behavior. Step fusion serializes dependencies between steps, eliminating synchronization overhead caused from obligatory *put* and *get* calls. For steps with common consumer dependencies, fusing those steps reduces the total memory bandwidth during runtime. In LULESH, tag tiling also reduces total data communication required by step computations when neighboring data is local to a tile, and there is possible data reuse between neighbors. However, these optimizations require moderate changes to the step routines.

3.4 Data Tiling

Following task coarsening through tag tiling and step fusion, we can perform data tiling optimizations to coarsen data in the item collection. Although the

total number of algorithmic steps is reduced along with the number of tags prescribing those steps, the data elements are singleton values dynamically assigned by the CnC runtime, requiring a multitude of *gets* for each element or node dependency in the tiled step. Although straightforward, revamping the data layout of a program is a time consuming task, and potentially prohibitive depending on the specific application. For LULESH, we modify kernel routines and place calls inside CnC steps which provide flexible parameters and future modifications.

Modifications to core computations aim to take advantage of data locality and reduce communication using larger block sizes. We create tiled objects and use pointers to reduce unnecessary data movement. However, the data is treated as immutable, using *get/put* clauses to ensure proper synchronization and execution. During the node-to-element force computation, we overlap node tiles at element interfaces, propagating communications across tiles in a wave front manner, removing the need for two-way communication to update both tiles. Spatial stencil computation is also optimized and packed to match tile-size, requiring additional code changes. Despite underlying code changes, performance benefits from data tiling cannot be overlooked, especially in LULESH where numerous data items are used at every node/element and sometimes persist for multiple iterations.

We note that without first performing tag tiling, and ideally step fusion, data-tiling is not a viable optimization. Without coarsened tasks, blocked data is not useful under the strict dynamic-single-assignment properties of CnC item collections. In our experiments, we compare this final full-tiled implementation of LULESH to our other progressions as well as OpenMP implementations distributed by LLNL.

4 Results

In this section, we evaluate the performance of our multiple configurations of the CnC LULESH application for a problem size of 60^3. These include the domain expert baseline, a fused-only, a tiled-only, a fused&tiled, and a fully-tiled implementation. Additionally, we benchmark the LLNL LULESH 2.0.3 implementation with OpenMP directives as a comparison representing a more traditional parallel programming model. We measure their execution times running on our shared-memory system running on up to 48 processors. The following implementations are tested:

Baseline - Our baseline expresses the LULESH application at its most decomposed level, with minimal dependence constraints. There are 13 steps, 35 data items, and 3 tags which prescribe steps for every iteration, node, and element in the mesh, requiring dynamic step instances for each, but allow any order of scheduling. The item collections also correspond to individual nodes and elements in the mesh. It follows CnCs principles of expressing a program as partially ordered computations and its dependencies, but excessive fine-grained parallelism plagues performance.

Fused only - Using step fusion, we reduce the step collection size from 13 to 5. This minimizes the number of prescribed dynamic steps as well as several

consumer/producer data dependencies, reducing the item collection size by 5. However, communication and scheduling overheads prevent scaling.

Tiled only - Tiling coarsens the tag space by prescribing blocks of work corresponding to a 3-dimensional spatial block instead of individual element, improving scalability and performance by reducing scheduling overhead and improving data locality. A tilesize of 10-15 is typically used for a problem size of 60 when running on 48 processors. The CnC specification is identical to the baseline.

Tiled & Fused - Both step fusion and tag tiling are applied at a high level. In step routines, we attempt to exploit locality for data that is shared between common neighbors, as well as reuse common data inputs from fused steps. These transformations require some coding changes and extra bookkeeping for extra variables and computation re-ordering to preserve step-like properties required by every CnC step. The corresponding CnC specification contains 5 steps, 27 data items, and 3 tags which prescribe steps for every tiled block. However, the data items still pertain to individual elements and nodes.

Data Tiled - The data tiled code incorporates the optimizations from step fusion and tag tiling, as well as tailoring each task with its working data set. A single *get* and *put* reads or writes a block of variables for each tiled computation step, albeit most steps still require multiple *gets* due to needing multiple data sets from different sources. The underlying computations are rewritten to accommodate the updated data structures. There are still 5 steps, 27 data items, and 3 tags, but data items are of a *tiled* construct.

4.1 Evaluation

Experiments were run on mesh sizes up to size 60^3 for 30 iterations, ten times per configuration, 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) per socket, each processor 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 for each configuration.

Figure 4 shows the performance speedups against the sequential baseline for our 4 benchmarks of LULESH in CnC and the provided OpenMP code from LLNL. For our CnC baseline, 60^3 dynamic step instances are created for each minimally-constrained step, performing and scaling extremely poorly. Applying step fusion reduces the number of steps by more than half, and results in a 1.6-2.5x speedup, with some improvement in parallel execution. Fusion by itself does not impact when compared to tiling, which coarsens the computation to a much greater extent. Looking at the tiled only implementation, we see speedups of 60x compared to sequential baseline when running on 48 threads. This improvement is a result of coarser grained steps, reducing the synchronization required by the scheduler to instantiate the schedule so many step instances. In our next code iteration, we combine both step fusion and tag tiling technique, yielding greater

Table 1. LULESH Iteration Runtimes (sec): 60^3 Sized Mesh

	Number of Cores						
	1	**2**	**4**	**8**	**16**	**32**	**48**
Baseline	148.40	141.68	135.18	154.89	160.27	154.70	158.47
Fused Only	101.36	95.281	72.273	58.508	60.269	59.995	64.056
Tiled Only	19.147	18.919	10.539	5.7492	3.4986	2.4606	2.2643
Tiled&Fused	11.767	11.725	6.5347	3.9041	2.3639	1.6201	1.3920
Data Tiled	0.2268	0.2339	0.1242	0.0644	0.0360	0.0255	0.0277
OpenMp	0.6882	0.3784	0.2167	0.1219	0.0852	0.0814	0.0833

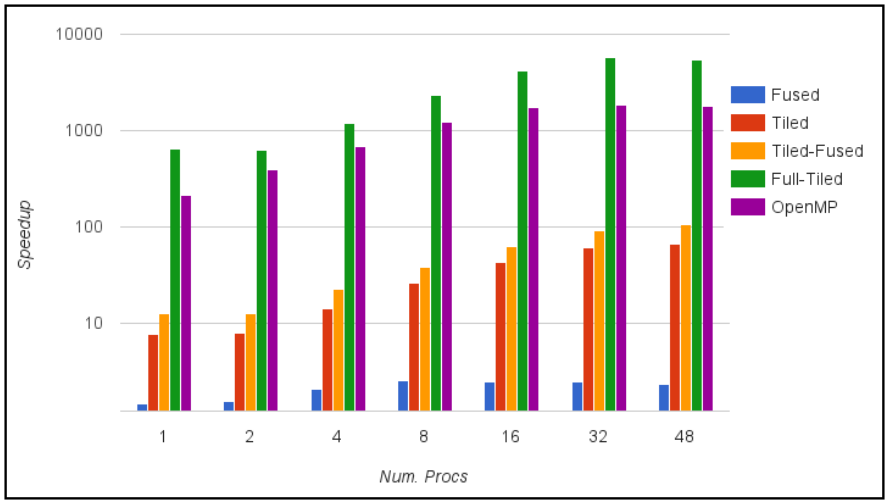

Fig. 4. Performance Speedup

performance, but it still does not surpass the performance from OpenMP. Finally, our fully tiled LULESH code with step and data tiling gives an additional order of magnitude of performance improvement over purely task coarsening implementations (note logarithmic axis). Tiling the data collections to correspond to the step collections When compared to similar processor configurations in OpenMP, our CnC code outperforms it by 3x for 32 and 48 processor. We reason that the OpenMP implementation has a number of inefficiencies, such as requiring barriers before each ghost exchange, as well as extra data movement to temporary buffers when updating data for reduction operators using multiple threads. Because CnC utilizes an asynchronous task-parallel model, it is more efficient than synchronous models such as OpenMP. However, both programs perform the exact same computation—the difference being the scheduling of work and movement of data.

In Figure 5 we observe almost no scaling from the non-tiled implementations, whereas the tiled codes exhibit weak scaling, starting at 4 processors. How-

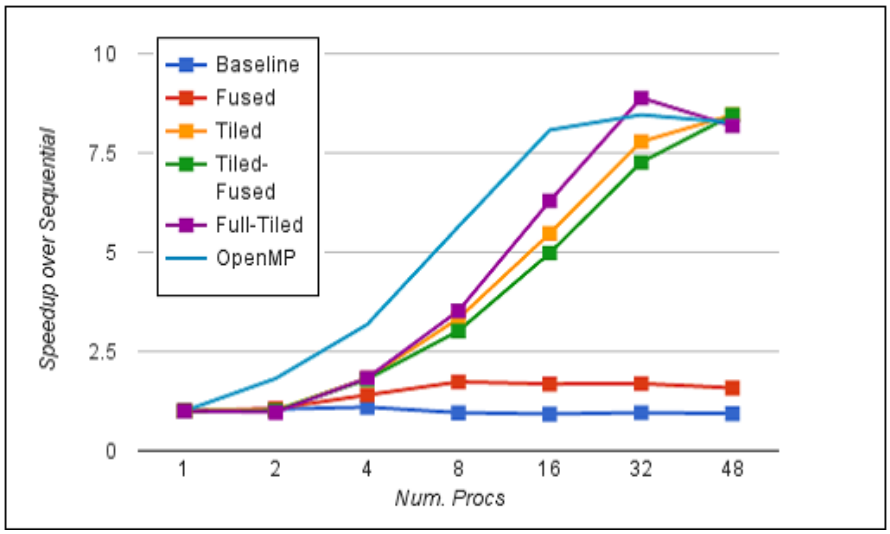

Fig. 5. Scalability Results

ever, CnC dedicates one processor exclusively for scheduling purposes, whereas OpenMP does not since parallelism must be explicitly expressed by the programmer. As a result, OpenMP offers an advantage when a few processors are used, but our fully-tiled code scales more strongly. Scaling beyond 32 processors should be possible, but we reason our machine configuration skews results at 48 cores. In the next section, we discuss the lessons learned and recommend an approach for achieving high performance while maintaining programmability.

5 Lessons Learned

In our study, we focus on the LULESH application, starting from the domain expert's minimally constrained algorithm, and applied high level fusion and tiling transformations on the program by altering the step, data, and tag collections while preserving program semantics. However, the applicability of these coarsening techniques are not limited to LULESH. Multiple factors contribute to performance improvement over the baseline LULESH code. From the perspective of code modifications, step fusion requires the fewest modifications, while data tiling requires an overhaul of underlying data structures and computation code. Both step fusion and tag tiling give substantial speedup, with tag tiling provides the most benefit, but it was a prerequisite for implementing data tiling in our application. Once the cohesive tiling implementation was produced, the performance of LULESH using CnC begins to shine and greatly outperforms the OpenMP implementation.

In hindsight, the most efficient method would have been to decompose the algorithm, compose the computation steps for generalized tiled data, and then map those computations to a high level *domain specification* that can be mapped to a valid *CnC specification*. Such a process would generate similar results to

our final implementation while providing flexibility to apply step fusion and tag tiling for various tile sizes. We recommend using the *CnC translator* to generate source code containing the *CnC context* and additional scaffolding step code from the high level specification. This translator was recently developed by the CnC Habenero research group to assist their work on declarative tuning [6]. However, it is not a tuning mechanism, but an automation tool provided for programming portability. Following their syntax to describe the *CnC specification*, which include all tags collections, item collections, steps, and their dependencies, source files will be generated that for the context as well as skeleton code for each step with predetermined *get* and *put* constructs. The programmer's primary responsibility is to initialize their problem, set up their work routines, and insert the proper computation for each step. In our final tiled LULESH implementation, the CnC code and work routines were decoupled in such a way. Using this translator along with modular kernel routines, while keeping granularity in mind, should improve productivity while preserving performance for future CnC applications.

6 Related Work

Parallelizing applications requires programmers to be keenly aware of a range of system level as well as algorithmic details in order to achieve performance speedup. Managing this level of detail remains a difficult task, even for the most experienced programmers. In addition, determining the best trade-off between programming portability and performance is an active research area. Concurrent Collections is just one approach that uses a model that takes advantage of asynchrony and task-based parallelism to efficiently program parallel applications.

Task Parallel Models. Researchers have begun to shift toward task-parallel and asynchronous many-task models to provide performance portability for high performance scientific applications. In recent years, programming models such as CnC, Charm++, Legion, OpenMP 4.0 have began a trend toward programmability with task-parallel support, but none have matured into a one-size-fits all solution [11, 12, 3]. Legion avoids employing data-drive execution and instead focuses on controlling execution via mapping interfaces, opposite to the CnC approach. Charm++ offers similar constructs to CnC, but uses a message passing interface for driving execution and offers fewer high level abstractions. OpenMP has long been a recognized for its superior ease of parallel programmability, but has only recently supported task-based parallelism. In our work, we show our CnC tuned version of LULESH greatly outperforms older models such as OpenMP, but we surmise performance would at least rival those of newer task-based models.

Tiling. Although tiling is a well-known technique, there are few practical ways to obtain automatically tiled code. Researchers have long tried to obtain coarse grained task parallelism since the early versions of OpenMP [13]. Other approaches have employed polyhedral frameworks such as PLuTo to generate tile

loop iterations for matrix based computations, as seen in Kong et al. [14, 15]. However, their approach creates coarsened computation for affine loops contained matrix computations, unlike LULESH, which requires irregular control flow using data dependencies from various computation methods. 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. utilizes the CnC specification to produce tiled code, but those tiles leverage OpenMP directives to achieve parallelism.

7 Conclusion

In this paper, we discuss and evaluate the performance impact of coarsening the step, tag, and data collections of the LULESH written in CnC. Although Concurrent Collections offers intuitive parallel programming constructs, achieving good performance requires program tuning that does not directly follow the separation of concerns philosophy. In our work, we demonstrate the effects of task and step coarsening to improve the performance and scalability of the LULESH application. We begin with a decomposed LULESH algorithm consisting of minimally constrained computational steps. Step fusion and tag tiling optimizations improve performance by coarsening the task-granularity of the program, and creates the opportunity to additionally tile the data collection to reduce data synchronization overheads. This fully tiled CnC LULESH code outperforms OpenMP parallel implementations by 3x for up to 48 processors and exhibits scalable performance. In our discussion, we present the CnC translator as a means of generating CnC code to handle control flow and data synchronization between steps. In the future, we hope to extend the functionality of the translator tool as well as provide better abstractions for handling task and data coarsening in CnC.

CnC goes beyond just scientific applications. The CnC philosophy to approach algorithms using collections is aimed to abstract layers of complexity of hardware mapping and work scheduling at the thread level. Dedicated tuners exist for that purpose of optimizing platform-specific hardware, but our contribution is to identify the ideal CnC code to run on those machines. Naive programmers will be quick to discredit the merits of CnC when they believe the ease of programmability comes at the price of poor performance when their application is minimally constrained. Instead, using the available CnC translator and an approach that takes task granularity in mind, one can achieve both programmability and performance in CnC.

Acknowledgments. This research is supported by the Department of Energy under contract DE-FC02-12ER26104. We would also like to thank Ellen Porter, Kath Knobe, Nick Vrvilo, and Zoran Budimlic for their comments and feedback during discussions regarding CnC.

Bibliography

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

[2] Karlin, I., Keasler, J., Neely, R.: Lulesh 2.0 updates and changes. Livermore, CAAugust (2013)

[3] OpenMP, C.: C++ application program interface (2002)

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

[5] Burke, M.G., Knobe, K., Newton, R., Sarkar, V.: Concurrent collections programming model. In: Encyclopedia of Parallel Computing. Springer (2011) 364–371

[6] Chatterjee, S., Vrvilo, N., Budimlić, Z., Knobe, K., Sarkar, V.: Declarative Tuning for Locality in Parallel Programs. In: Proceedings of the 45th International Conference on Parallel Processing. ICPP '16 (August 2016) To appear.

[7] Sbîrlea, A., Zou, Y., Budimlíc, Z., Cong, J., Sarkar, V.: Mapping a data-flow programming model onto heterogeneous platforms. In: ACM SIGPLAN Notices. Volume 47., ACM (2012) 61–70

[8] Habanero-Rice: Concurrent collections on ocr (2015)

[9] Frank Schlimbach, I.C.: Intel concurrent collections for c++ for windows and linux (2015)

[10] Liu, C., Kulkarni, M.: Optimizing the lulesh stencil code using concurrent collections. In: Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing, ACM (2015) 5

[11] Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: expressing locality and independence with logical regions. In: Proceedings of the international conference on high performance computing, networking, storage and analysis, IEEE Computer Society Press (2012) 66

[12] Kale, L.V., Krishnan, S.: CHARM++: a portable concurrent object oriented system based on C++. Volume 28. ACM (1993)

[13] Kasahara, H., Obata, M., Ishizaka, K.: Automatic coarse grain task parallel processing on smp using openmp. In: International Workshop on Languages and Compilers for Parallel Computing, Springer (2000) 189–207

[14] Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: 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)

[15] Kong, M., Pop, A., Pouchet, L.N., Govindarajan, R., Cohen, A., Sadayappan, P.: Compiler/runtime framework for dynamic dataflow parallelization of tiled

programs. ACM Trans. Archit. Code Optim. **11**(4) (January 2015) 61:1–61:30

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