Declarative Tuning for Locality in Parallel Programs

Sanjay Chatterjee, Nick Vrvilo, Zoran Budimlić, Kathleen Knobe, Vivek Sarkar
Department of Computer Science
Rice University, Houston, TX 77025 USA
sanjay.chatterjee@gmail.com, {*nick.vrvilo, zoran, kath.knobe, vsarkar*}*@rice.edu*

***Abstract*—Optimized placement of data and computation for locality is critical for improving performance and reducing energy consumption on modern computing systems. However, for most programming models, modifying data and computation placements typically requires rewriting large portions of the application, thereby posing a huge performance portability challenge in today's rapidly evolving architecture landscape. In this paper we present TunedCnC, a novel, declarative and flexible CnC tuning framework for controlling the spatial and temporal placement of data and computation by specifying hierarchical affinity groups and distribution functions. TunedCnC emphasizes a separation of concerns: the domain expert specifies a parallel application by defining data and control dependences, while the tuning expert specifies how the application should be executed on a given architecture—defining *when* and *where* for data and computation placement. The application remains unchanged when tuned for a different platform or towards different performance goals. We evaluate the utility of TunedCnC on several applications, and demonstrate that varying the tuning specification can have a significant impact on an application's performance. Our evaluation is performed using an implementation of the Concurrent Collections (CnC) declarative parallel programming model, but our results should be applicable to tuning of other data-flow task-parallel programming models as well.**

***Keywords*—declarative, locality, tuning, scheduling, tasks**

I. Introduction

With the recent explosion in the amount of homogeneous and heterogeneous parallelism available in the hardware (e.g., vector, multicore, SMPs, GPGPUs, FPGAs, DSPs, clusters), as well as increasing complexity in memory hierarchies (e.g., scalar/vector registers, scratchpad memories, on-chip caches, off-chip caches, high-bandwidth memories), it is becoming increasingly difficult, even for experts, to tune application performance for different machines while also striving for performance portability. In particular, tuning parallel applications for improved locality is a very significant challenge because of the inherent trade-offs between parallelism and locality, which can be manifest in different ways on different hardware platforms.

Recently, there has been an emergence of higher-level programming systems that reduce the burden of parallel programming. Asynchronous task-parallel programming models, such as those available in Chapel [1], Cilk [2], Concurrent Collections (CnC) [3], Habanero-C (HC) [4], Habanero Java (HJ) [5], Legion [6], OpenMP 4.0 [7], and X10 [8] enable programmers to decompose their algorithms into tasks, and offer high-level synchronization mechanisms to coordinate task execution with improved correctness and performance guarantees compared to programming with threads and locks.

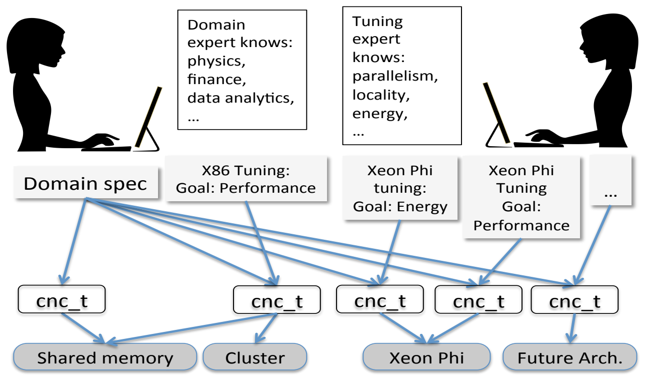
Domain expert knows: physics, finance, data analytics, ...
Tuning expert knows: parallelism, locality, energy, ...
Domain spec
X86 Tuning: Goal: Performance
Xeon Phi tuning: Goal: Energy
Xeon Phi Tuning Goal: Performance
...
cnc_t
cnc_t
cnc_t
cnc_t
cnc_t
Shared memory
Cluster
Xeon Phi
Future Arch.

Fig. 1: A *separation of concerns* between domain and tuning experts. Domain expert creates a single domain spec for the application. Tuning expert creates different tuning specs depending on the target platform and the tuning goals (performance, energy, resilience, etc.). The system creates a tuned application for a given platform and tuning goal.

Some of these models (e.g., Legion, OpenMP 4.0) allow inter-task dependences to be specified implicitly using the task superscalar model [9], whereas others (e.g., CnC, HC, HJ) allow task dependences to be specified explicitly using the data-driven task model [10].

In this paper, we present TunedCnC, a novel, declarative and flexible tuning framework focused on a *separation of concerns*. The program's dependence graph (*domain spec*), indicates the computation ordering requirements and leaves tuning decisions to an optional *tuning spec*, as shown in Fig. 1. Because of this separation, the same domain spec (without modification) can easily be paired with different tunings (labeled `cnc_t` in Fig. 1). In our approach, the domain expert implements an algorithm by decomposing it into *computation steps* and then constructing a graph of data and control dependences. The graph does not indicate anything about where or when the computations occur. A step can execute whenever its input data is available and control indicates that it should run.

The focus of this paper is on a new tuning approach designed to improve locality, using *hierarchical affinity groups* and *distribution functions*. Our model supports tuning for spatial and temporal locality on both shared and distributed memory systems. Section II summarizes the programming and tuning models assumed in our work, and Section III describes their implementation details. Section IV contains our experimental results, followed by a discussion of related work in Section V and our conclusions in Section VI.

II. Programming Model

The programming model for our approach includes a *domain spec* that can be written by a domain expert, and a *tuning spec* that can be written by a tuning expert. Consistent with our goal of separation of concerns, the tuning process is isolated from the development of the application, allowing for multiple tuning specs to be written for the same domain spec.

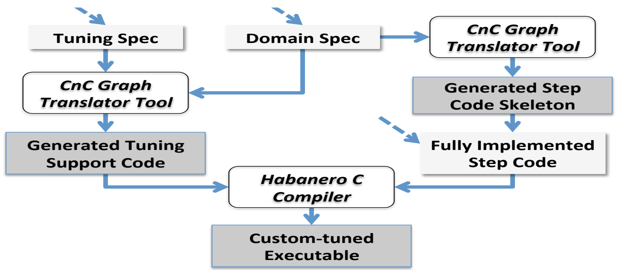
Tuning Spec
Domain Spec
CnC Graph Translator Tool
CnC Graph Translator Tool
Generated Step Code Skeleton
Generated Tuning Support Code
Fully Implemented Step Code
Habanero C Compiler
Custom-tuned Executable

Fig. 2: Typical workflow for implementing and tuning an application in TunedCnC. Dashed arrows indicate artifacts provided by the domain or tuning experts.

Fig. 2 shows a typical workflow of implementing and tuning an application in our system. The domain spec describes the flow of control and data throughout the application. This spec is fed through the graph translator tool to produce skeleton code for the application's computation steps. The domain expert also needs to implement the individual computation steps. The tuning expert writes a tuning spec for a given domain spec, hardware platform and tuning goal. The tuning and domain specs are also processed by the graph translator tool to generate the tuned support code for the application. The tuned support code and the implemented step code are then processed to generate a customized binary for executing on the target platform (the Habanero-C compiler is used for this purpose in our implementation.)

The Domain Specification: The domain specification indicates exactly and only the division of the application into tasks and the dependence constraints among those tasks. As an example, Fig. 3 shows the domain specification for a Cholesky factorization program.

```
($init:) → (cholesky:0...N)

(cholesky:iter) → (trisolve:k+1...N, iter)
(cholesky:iter) ← [array: iter, iter, iter]
(cholesky:iter) → [array: iter, iter, iter+1]

(trisolve:row, iter) → (update:iter+1...row+1, row, iter)
(trisolve:row, iter) ← [array: row, iter, iter]
(trisolve:row, iter) ← [array: iter, iter, iter+1]
(trisolve:row, iter) → [array: row, iter, iter+1]

(update:col, row, iter) ← [array: col, row, iter]
(update:col, row, iter) ← [array: col, iter, iter+1]
(update:col, row, iter) ← [array: row, iter, iter+1]
(update:col, row, iter) → [array: col, row, iter+1]
```

Fig. 3: Domain spec for Cholesky factorization. `$init` is a special kind of computation step that gets executed at the beginning of the program.

The notation for our domain specs is drawn from CnC. A domain spec contains nodes that correspond to computation steps. In Cholesky, there are three distinct computation steps: `(cholesky)`, `(trisolve)` and `(update)`. We use parentheses to denote step collections. Each of these is called a step *collection* in that it corresponds to a set of dynamic step instances. Step instances are distinguished by the value of their tags: `(cholesky:iter)`, `(trisolve:row,iter)` and `(update:col,row,iter)`.

The domain spec also has nodes that correspond to item collections, which is how data is represented in CnC. We use square brackets to denote item collections, which also use tags to distinguish item instances. In the case of Cholesky, there is only one item collection: `[array: col,row,iter]`.

Steps can have *producer* relationships (→) with other step/item instances, and *consumer* relationships (←) with item instances. These dependence constraints among the dynamic instances form a computation DAG. These relationships in the domain spec give exactly the constraints necessary to ensure correct execution of the program. See [3] for more details on the domain specification.

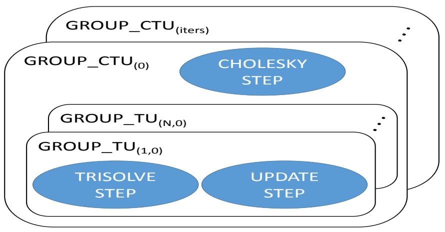

Fig. 4: Cholesky Affinity Groups

The Tuning Specification: We continue to use the Cholesky factorization example to illustrate our tuning language. The tuning language supports two types of declarations: *affinity groupings* and *distribution functions*. First we show how the computation can be organized into affinity groups, and then we show how to declare a custom distribution for data and computation.

A tuning spec, which contains a specific tuning of the application written using the tuning language, refers to objects from the domain spec. It is similar to the domain spec in style. Inconsistencies and conflicts between the domain and tuning specs are detected and flagged by the graph translator tool.

There are many possible affinity groupings for the Cholesky domain spec in Fig. 3. One such grouping is depicted in Fig. 4. Here we will assign a tight affinity between `(trisolve:row, iter)` and `(update:col, row, iter)`, by creating an affinity group, let's call it `TU`, that contains both of these steps:

```
(TU) → (trisolve), (update);
```

Notice that we don't want `(trisolve:4, 12)` to have an affinity with `(update:3, 28, 52)`. We want an affinity among instances that have the same value of `row` and `iter`. This means that there will be multiple dynamic instances of this group, one for each `row/iter` pair in a given run. This is achieved by creating groups in the tuning spec, with dynamic instances identified by *tags*. In this case the instances of group TU are exactly identified by a tag composed of row and iter. Now we have:

```
(TU:row, iter) → (trisolve:row, iter),
                 (update:0...iter, row, iter);
```

This means that for each row/iter pair in (TU:row, iter) we will have a dynamic instance of an affinity group enabling locality among the step instances within it. Above this in the affinity group hierarchy we will have a group called CTU that will have an instance per iteration as follows:

```
(CTU:iter) → (cholesky:iter),
             (TU:0...iter, iter);
```

Notice that this creates a higher-level affinity group that contains a single step instance together with a set of lower level affinity groups. Some groupings might be difficult to express in terms of an existing step's tag components. For that case, we support declaring an alias for an existing step collection, where the alias step's tag components are mapped onto the actual steps tag. Fig. 10 in Section IV demonstrates this functionality.

To achieve maximal parallelism, affinity groups, steps, and items must be distributed across the available compute resources. The special variable $RANKS represents the number of compute resources to target from the current level in the affinity group hierarchy. Section III describes the tuning execution model and implementation details. Items are always distributed at the highest level of the hierarchy, but affinity groups and steps are distributed at the hierarchy level corresponding to their nesting depth in the affinity groupings. The following declaration is equivalent to the default distribution function for array (cyclic in the iter dimension):

```
[array: col,row,iter]: { iter % $RANKS };
```

Fig. 7 in Section IV shows the full spec for an alternative tuning of the Cholesky factorization example.

III. Runtime Implementation

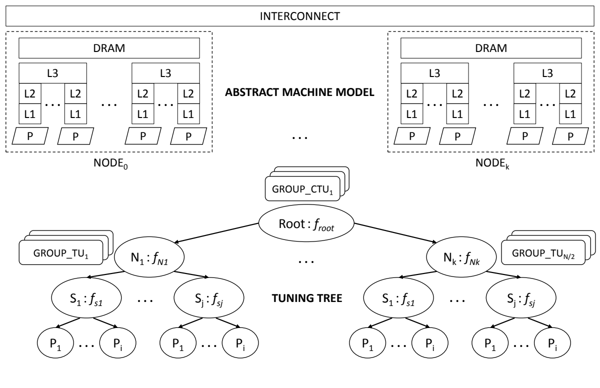

Fig. 5: Tuning tree abstraction for machine model

In our tuning framework, we use the concept of the *tuning tree* to abstract the underlying machine architecture, as shown in Fig. 5. The tree represents a structured hierarchical grouping of compute resources derived from the memory level hierarchy in the system. While leaf nodes represent the individual compute resources, the internal nodes are symbolic of a grouping of the compute resources in that sub-tree. The tuning framework exposes a special variable, $RANKS, to represent the number of compute resources under each node in the hierarchy. For example, on a distributed system, the root of the tuning tree may represent a logical grouping of all the coherent address space domains in the system—such as node-level MPI ranks—whereas lower levels might represent the number of sockets on a board, or the number of cores in a socket. The tuning tree configuration is provided by the tuning expert as an input file to the runtime during execution.

The ability to treat a cluster as another level in the platform hierarchy gives the tuning expert a tool to approach tuning programs for shared-memory and distributed-memory machines in the same manner. The domain expert's code remains unchanged regardless of whether the program is being executed on a shared-memory or distributed-memory machine, and regardless of what kind of tuning is applied to it.

The structure of the tuning tree and the hierarchical affinity groupings are important tools for the tuning expert to reason about the spatial and temporal locality of domain tasks. The affinity groupings and their distribution functions in the tuning spec manifest as specialized tuning tasks in the TunedCnC runtime. The CnC translator (shown in Fig. 2) generates the appropriate tuning tasks, which are used to assert the tuning spec semantics during runtime execution through the addition of extra dependences in the program. Using these specialized tuning tasks, the tuning runtime can now dynamically guide the placement (spatial) and staging (temporal) of the execution of domain tasks.

The spatial placement of tasks is handled by applying a distribution function to the task's affinity group at runtime. This may unpack the affinity group and distribute its elements, according to the distribution function, among the children of the current tuning tree node. By default, the runtime distributes cyclically based on the final component value of an instance's tag. Domain tasks that result from the unpacking of affinity groups get released to the domain runtime for execution.

Dependences on tuning tasks specified through the tuning spec enable the staging of distribution of affinity groups. This, along with the hierarchical affinity grouping, allows the tuning expert to affect the temporal locality along with the spatial placement through distribution functions. Logically, each node in the tuning tree holds a queue of affinity groups. When the tuning process starts, the highest-level affinity groups are on the queue at the top-most tuning tree node. In general, the process moves affinity groups down the tree by removing the group at the bottom of the queue at some tuning tree node, breaking the group up into its lower-level components and distributing these components to the children of this node, putting components at the top of each child's queue. This is illustrated in Fig. 5, which shows the distribution of the CTU affinity groups to create the TU groups on the child nodes.

For the results reported in this paper, we developed a new CnC graph translator tool to automatically convert the declarative domain and tuning specifications into Habanero-C source code for creating domain tasks, along with extensions for creating tuning tasks. The translator accepts as input a domain specification file and an optional tuning specification file. The translator generates skeleton code for the CnC step functions and support code to translate CnC operations into Habanero-C operations. After the domain expert fleshes out the generated skeleton code, the application can be compiled by the Habanero-C compiler. If a tuning specification is provided, then a tuning *actions* file is also generated to control the creation and management of tuning tasks. The tuning code adds constraints to the generated application on where and when the tasks are executed and where the data is located with respect to the tuning groups declared in the tuning specification.

The distributed-memory runtime system used in this work builds on the communication runtime introduced in the HCMPI [4] project, with new extensions for enabling a Hierarchical Place Tree (HPT) abstraction across a distributed-memory cluster and for supporting both tuning and domain actions in the same runtime system (thereby creating a new set of mechanisms for controlling the placement of data and computation within a distributed memory system). In contrast, past work on Hierarchical Place Trees (HPTs) [11] in the Habanero project only supported intra-node parallelism, and did not have any support for tuning actions.

IV. Experimental Results

In this section, we present the results of our experiments using the CnC tuning framework. We compare some tuning approaches for the tiled Cholesky factorization code, which has been our running example in this paper. We also show some tuning opportunities with the SmithWaterman and RicianDenoising examples. Benchmark and toolchain source code is available on GitHub.[1]

Our experiments were conducted on a cluster consisting of 16-core Intel Xeon CPU nodes. We report results for up to 16 nodes (256 cores). Each node consists of two sockets with each socket containing 8 cores and a shared L3 cache of size 20MB. Every core has private L1 and L2 caches of sizes 32K and 256K respectively. The cluster is networked through a 56 Gb/sec (4X FDR) Infiniband layer. On this system, our runtime communication layer was configured to use Intel's MPI library version 5.0.

Tuning Cholesky Factorization: Cholesky decomposition is a dense linear algebra application. This implementation of the benchmark runs a tiled version [10]. Each tile on every iteration is executed by a unique task. The computations within an iteration comprise of the *Cholesky* step on the diagonal pivot, followed by *trisolve* steps on the pivot column panel and *update* steps on the remaining tiles of the lower triangle. Within an iteration, the only data reuse is that of *trisolve* tiles by the *update* tiles. So, our strategy for exploiting data reuse is across iterations, since the *update* tiles are reused in the subsequent iteration.

[1]https://github.com/habanero-rice/cnc-ocr/tree/icpp2016-tuned-cnc

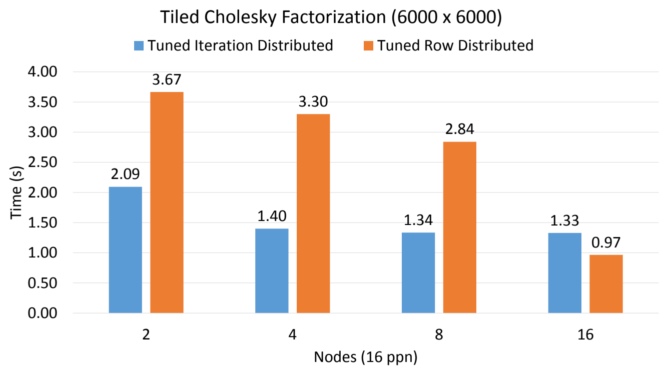

Fig. 6: Execution times for tuned distributed variations of a 6000×6000 Cholesky factorization. (Shared-mem single-node time was 2.74s.)

```
// item collection distributions
[array:j,i,k] : { k % $RANKS }; // same as default
[results:i] : { 0 };

// affinity group distributions
(CTU:k): { k % $RANKS }; // same as default

// affinity groups
($init:) → (CTU:0...#numTiles);
(CTU:k ) → (cholesky:k), (TU:k+1...#numTiles,k);
(TU:j,k) → (trisolve:j,k), (update:k+1...j+1,j,k);
```

Fig. 7: Cholesky iteration-wise tuning. The `#var` syntax denotes a global parameter of the CnC graph.

In Fig. 6, we show the results of two kinds of tuning on the distributed system, *iteration-distributed* and *row-distributed*, for a 6000×6000 Cholesky factorization. The baseline CnC version runs on a single node with 16 processors as a shared memory application with an execution time of 2.74 secs. The single-node OpenMP version of Cholesky for the same data size executed in 3.73 seconds on 16 cores.

The *iteration-distributed* tuning approach in Fig. 7 partitions the computation space into the different iterations of the Cholesky factorization. Each iteration is mapped to a specific node on the cluster. With this approach, all the steps in one iteration can benefit from node locality, while we allow parallelism by enabling tasks in successive iterations to start executing as soon as their dependences are met by the previous iteration. The *row-distributed* tuning approach partitions the rows and map them to specific nodes. So, all tasks in a row that gets executed for multiple iterations get mapped to the same node. This enables inter-iteration reuse. It is interesting to note that while tuning with row distribution initially performs worse than tuning with iteration distribution, it eventually scales better. This result reaffirms the need for flexible tuning mechanisms to take advantage of variability in platform configurations, and to allow for different tunings at different scales of parallelism.

Tuning Smith-Waterman: In this section we describe our experiences with tuning Smith-Waterman kernel, a frequently used sequence alignment algorithm in biomedical applications.

In our experiment, we used a tiled version of the Smith-Waterman algorithm [4] over sequences of length 185.6k and 192k. Our tiles contained 928×960 elements each. Each tile computation depends on values from its above, left and upper-left neighbors. This dependence pattern enables Smith-Waterman to progress with an asynchronous computation wavefront. We use this insight into the algorithm to explore three different tuning strategies: *wave distributed*, *row distributed*, and *col distributed*. These tuning approaches use only distribution functions, without using any affinity groups. Fig. 8 shows the performance of these three approaches.

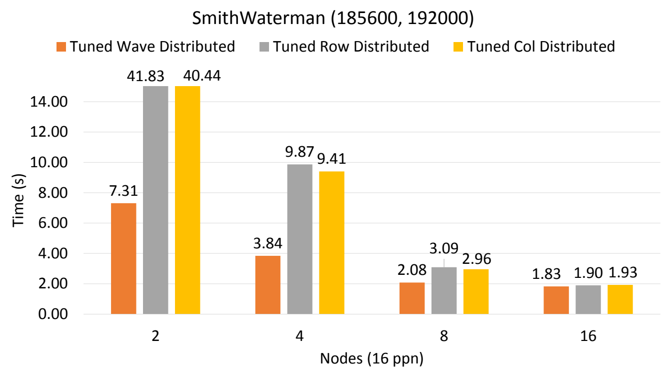

Fig. 8: Execution times for tuned distributed variations of SmithWaterman with string lengths of 185,600 and 192,000. (Shared-mem single-node time was 10.35s.)

The *wave distributed* approach aims to enable locality, without reducing parallelism, by chunking each diagonal. Neighboring tiles in a chunk get mapped to the same node. Fig. 9 shows the actual tuning code that maps the nodes on the wavefront. The step computations, performed by *swStep*, are distributed across the system by a tuning function called *wave_dist*, which is a user-defined distribution function that the tuning expert writes and links with the application. It is called from the tuning code. The *row/col distributed* approaches distribute the rows/columns across the nodes in cyclic manner. These results demonstrate that the most effective tuning might not be easily expressible in our tuning DSL, confirming the need to support calls external user-defined functions from within a tuning specification.

Tuning Rician Denoising: The 2D Rician Denoising application is part of a medical imaging pipeline developed in the NSF Expeditions CDSC research project [12]. The application performs five-point stencil computation on each pixel of an image. The computation is done iteratively until convergence. The baseline task-parallel implementation of the application performs the stencil computation on a 2D tile of elements, while executing all tiles of an iteration in parallel.

```
// item collection distributions
[above:i,j]: {wave_dist(i,j,#tilesPerRow,$RANKS)};
[left: i,j]: {wave_dist(i,j,#tilesPerRow,$RANKS)};

// step collection distributions
(initAboveStep:): { 0 };
(initLeftStep: ): { 0 };
(swStep:i,j): {wave_dist(i,j,#tilesPerRow,$RANKS)};
```

Fig. 9: Smith-Waterman wave-distribution tuning. The `#var` syntax denotes a global parameter of the CnC graph.

```
(rdTunedStep: iter, row, col, iter0, row0, col0 =
 rdStep: row0 + row, col0 + col, iter0 + iter);

(pyramid: row0, nRow, col0, nCol, nIter, iter0)
 → (rdTunedStep: iter @ 0...nIter,
                 iter...(nRow - iter),
                 iter...(nCol - iter),
                 iter0, row0, col0);
```

Fig. 10: Rician denoising pyramid tuning, showing a step mapping for reshaping `rdStep`'s tag, and the `pyramid` tuning group. The *var*`@`*expr* syntax gives a name to a tag component's value.

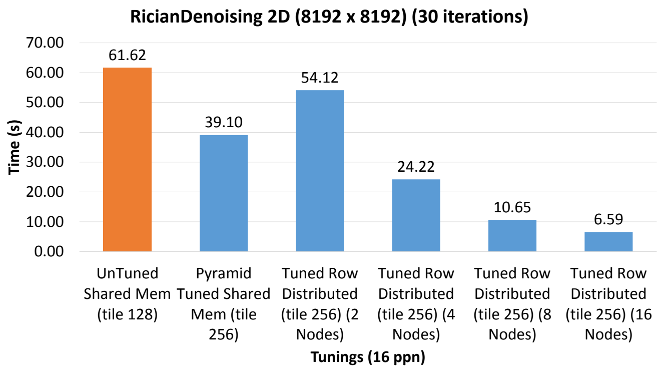

Fig. 11: Tuned Rician Denoising performance

We have developed both shared and distributed memory specific tunings. Our tuning strategy on the shared memory targets the reuse of the tiles computed from one iteration to the next as early as possible. This is to benefit from temporal locality of data in caches. Since the stencil computation needs to read the data from neighboring tiles, the area of computation in successive iterations that use the same tiles keep decreasing. Stacking these tiles across iterations produce the effect of constructing a pyramid. In order to benefit from tile reuse, the whole pyramid must fit in the socket level shared L3 cache. Our tuning orders the pyramids such that a new pyramid is brought into the shared L3 cache only when the current pyramid is close to being done. Fig. 10 shows the tuning spec that describes the affinity group for the pyramid. For the distributed memory, we use a simple tuning spec declaring a per-row distribution of the matrix. Fig. 11 shows the results of the performance of both tuned and untuned versions of the application. In our experiment, we use a matrix of size 8192 elements as the input. The best tile size of the untuned version was 128×128 elements. In contrast, a tile size of 256×256 elements performed best for the pyramid-tuned version. We observed that ordering the computation with the pyramid tuning performs significantly better than the untuned version. This result shows that tuning for temporal locality can have a significant impact on performance. On distributed nodes, we scaled the row-distributed version up to 16 nodes or 256 cores using a tile size of 256×256 elements.

V. Related Work

Legion Mappers [6] are user-defined callback functions that are called by the runtime when deciding where to map certain parts of data and computation. The mapper is a part of an application. In contrast, we propose a declarative approach that allows the tuning expert to control *when* the computation is happening, in addition to *where* the data and computation is placed through the use of affinity groups. This allows us to tune for *temporal locality* in addition to *spatial locality*, and to avoid interference from different affinity groups that should not be executing at the same time.

Past work on tuning for locality includes cache-oblivious algorithms, auto-tuners and domain specific stencil compilers. The Pochoir [13] and Halide [14] compilers target data reuse from trapezoidal computations, similar to the pyramidal structure in our tuning of Rician Denoising. The key difference is that the data reuse they target is specific to the cache hierarchy of each processor, while our work aims to benefit also from data reuse of the shared caches through runtime co-scheduling of tasks.

The approach we are describing in this paper is runtime-based, as opposed to compiler-based approaches proposed by autotuning [15], [16], [17]. This allows the tuner to use tuning actions to make more dynamic decisions reacting to the changes in the application behavior.

Recent work [18] has shown that dynamic selection of depth-first vs breadth-first scheduling decisions based on statically analyzed memory access patterns can significantly improve performance. In contrast, our framework allows the tuning expert to express any scheduling constraint for dynamic task graphs.

Many programming systems, such as X10 [8], Chapel [1] and Habanero-Java [5], [11] introduce a mechanism to specify *places* or *locales* to localize data and computation, but to the best of our knowledge, this work is the first to unify spatial and temporal locality using a declarative and dynamic approach.

VI. Conclusions

In this paper, we have presented a dynamic and declarative approach for controlling the locality of computation and data on shared and distributed memory systems for dependence-based execution models. A declarative tuning specification based on hierarchical affinity grouping of tasks in the Concurrent Collections (CnC) programming system enables user-directed task grouping and ordering for optimizing both spatial and temporal locality. We have implemented the runtime and compiler support that automatically translates the declarative tuning specification and combines it with the application into an executable that is tuned for a specific platform and for a specific tuning goal. This approach allows the domain expert to focus on the application (the data and control dependences in the algorithm), while at the same time allowing the tuning expert to affect the execution by constraining *how* data and computation should be grouped, *where* they should be located, and *when* should the computation tasks be executed.

We show that a choice of data distribution, computation distribution and computation ordering can have a great impact on application performance. We anticipate even larger impact on future systems with more cores, deeper hierarchies and higher ratios in the costs of data movement and communication vs computation.

Acknowledgments

This work was supported in part by the X-Stack program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (ASCR). We would like to thank Michael Burke for laying the foundations of the tuning language semantics and acknowledge his early work on the tuning language in [19].

References

[1] Cray Inc., "The Chapel language specification version 0.4," Cray Inc., Tech. Rep., Feb. 2005.

[2] R. D. Blumofe *et al.*, "CILK: An efficient multithreaded runtime system," *(PPoPP'95)*, pp. 207–216, Jul. 1995.

[3] Z. Budimlić *et al.*, "Concurrent Collections," *Scientific Programming*, vol. 18, pp. 203–217, August 2010.

[4] S. Chatterjee, S. Tasirlar, Z. Budimlić, V. Cavé, M. Chabbi, M. Grossman, Y. Yan, and V. Sarkar, "Integrating Asynchronous Task Parallelism with MPI," in *IPDPS '13*.

[5] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-Java: the New Adventures of Old X10," in *PPPJ'11*.

[6] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," ser. SC '12, Los Alamitos, CA, USA.

[7] OpenMP Architecture Review Board, *OpenMP Application Program Interface, Version 4.0*, July 2013, http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[8] P. Charles *et al.*, "X10: An Object-oriented Approach to Non-uniform Cluster Computing," ser. OOPSLA '05, pp. 519–538.

[9] Y. Etsion *et al.*, "Task superscalar: An out-of-order task pipeline," ser. MICRO '43. Washington, DC, USA: IEEE Computer Society, 2010, pp. 89–100, http://dx.doi.org/10.1109/MICRO.2010.13.

[10] S. Taşırlar and V. Sarkar, "Data-Driven Tasks and their Implementation," in *ICPP'11*, Sep 2011.

[11] Y. Yan *et al.*, "Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement," in *LCPC'09*, ser. Lecture Notes in Computer Science, vol. 5898, 2009.

[12] Center for Domain-Specific Computing, "Customizable Domain-Specific Computing," http://cdsc.ucla.edu.

[13] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The Pochoir Stencil Compiler," ser. SPAA '11, New York, NY, USA, 2011, pp. 117–128.

[14] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," ser. PLDI '13, 2013.

[15] K. Datta, "Auto-tuning Stencil Codes for Cache-Based Multicore Platforms," Ph.D. dissertation, EECS Department, University of California, Berkeley, Dec 2009.

[16] J. Shin *et al.*, "Speeding Up Nek5000 with Autotuning and Specialization," ser. ICS '10, 2010, pp. 253–262.

[17] T. Yuki, G. Gupta, D. Kim, T. Pathan, and S. Rajopadhye, "AlphaZ: A system for design space exploration in the polyhedral model," ser. LCPC '12, Sep. 2012.

[18] H.-S. Kim, I. El Hajj, J. Stratton, S. Lumetta, and W.-M. Hwu, "Locality-centric thread scheduling for bulk-synchronous programming models on cpu architectures," ser. CGO '15, 2015.

[19] K. Knobe and M. G. Burke, "The Tuning Language for Concurrent Collections," *16th Workshop on Compilers for Parallel Computing (CPC)*, 2012.