

# Applying the Concurrent Collections Programming Model to Asynchronous Parallel Dense Linear Algebra

Aparna Chandramowlishwaran

Georgia Institute of Technology  
aparna@gatech.edu

Kathleen Knobe

Intel Corporation  
kath.knobe@intel.com

Richard Vuduc

Georgia Institute of Technology  
richie@cc.gatech.edu

## Abstract

This poster is a case study on the application of a novel programming model, called Concurrent Collections (CnC), to the implementation of an asynchronous-parallel algorithm for computing the Cholesky factorization of dense matrices. In CnC, the programmer expresses her computation in terms of application-specific operations, partially-ordered by semantic scheduling constraints. We demonstrate the performance potential of CnC in this poster, by showing that our Cholesky implementation nearly matches or exceeds competing vendor-tuned codes and alternative programming models. We conclude that the CnC model is well-suited for expressing asynchronous-parallel algorithms on emerging multicore systems.

**Categories and Subject Descriptors** D.1.3 [Programming Techniques]: Concurrent Programming

**General Terms** Algorithms, Performance

## 1. Introduction

Motivated by multicore and future manycore architectures that require fine-grained thread- or task-level parallelism, researchers in the dense linear algebra community have been developing novel asynchronous-parallel algorithms [2]. In contrast to prior classical bulk-synchronous variants of such algorithms, these asynchronous variants (a) are naturally suited to cores with relatively smaller cache or local-store memories, and (b) reduce the degree of synchronization, whose cost may reasonably be expected to increase with increasing core counts. The current trend in research in this area is the pursuit of higher-level programming models to better support asynchronous algorithms, primarily using pragma and specialized library-based approaches [5, 6, 7].

In this poster, we study the feasibility of a novel general-purpose parallel programming model, called Concurrent Collections (CnC) [1], for dense linear algebra. We show that asynchronous variants of Cholesky factorization can be expressed in CnC, and when coupled with a well-tuned BLAS, closely match or sometimes exceed the performance and scalability of the Intel Math Kernel Library (MKL) implementation. Our achieved performance also compares favorably to PLASMA, a state-of-the-art domain-specific library-based approach.

## 2. Overview of CnC

CnC [1, 3, 4] is a novel programming model that separates the specification of computation from the expression of its parallelism. The goal is to simplify the task of a domain expert, who is responsible for designing the computation, from the tasks of a parallelization and tuning expert (or a compiler), who identifies the parallelism and performs scheduling/distribution and manages communication/synchronization. This section provides an overview of CnC concepts using a simple linear algebra example.

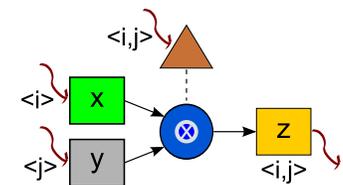
**Computation specification.** Consider the dense outer product computation,  $Z \leftarrow x \cdot y^T$ , where  $x$  and  $y$  are two column vectors and  $Z$  is a matrix of appropriate dimension. Algorithmically, we compute  $Z_{i,j} \leftarrow x_i \cdot y_j$  for all pairs,  $(x_i, y_j)$ .

The domain expert specifies the computation in a form that can be represented by a graph, as shown in Figure 1. This graph has 3 kinds of nodes: computational *steps*, data *items*, and control *tags*. Directed edges show producer-consumer relationships among these nodes. The graph is specified by the programmer using a formal textual representation.

A *step* is the basic unit of execution, which for the outer product is pairwise element multiplication. We consider this very fine granularity for example only, as in practice one might wish to choose a larger grain, such as a block. In the figure is a *step collection*, which statically represents the set (collection) of dynamic instances of these multiplications.

Data is represented using *item collections*. In this example,  $x$ ,  $y$ , and  $Z$  are the three item collections, denoted by boxes. These items serve as the basic unit of storage, communication, and synchronization.

Each instance of a step or item has an unique application-specific identifier, or *tag*. For the outer product, it is natural to use element indices as tags. We denote the tag for  $x$  by  $\langle i \rangle$ ,  $y$  by  $\langle j \rangle$ , and  $Z$  by  $\langle i, j \rangle$ . *Tag collections* (also called *tag spaces*) specify exactly which instances of a step will execute. A step collection is associated with exactly one tag collection; a step instance executes only if a matching tag instance exists. For the outer product, we show the tag space by a triangle and denote it by  $\langle i, j \rangle$ . For instance, only if the tag collection has  $\langle 3, 10 \rangle$  does the corresponding pairwise multiply for  $Z_{3,10} \leftarrow x_3 \cdot y_{10}$  execute. We say that a tag collection *prescribes* a step collection, and show that visually with a dashed undirected edge. Importantly, tags indicate *whether* a step will execute, but nothing about *when* it executes. This distinction



**Figure 1.** CnC graphical representation of the outer product operation,  $Z \leftarrow x \cdot y^T$ .

shows in part how CnC separates scheduling decisions from the computation’s specification.

Though not shown here, a step may produce tags. In this way, a step may control what other steps execute. This facility is part of what makes CnC a more flexible and general model than, say, a pure streaming language.

Lastly, the figure contains “squiggly” lines, which means the item or tag comes from or goes to the *environment*, which is the external code that invokes this computation. For the outer product, the environment provides the data items and control tags.

**Semantics and execution.** If a step executes and produces an item or a tag, that item or tag becomes *available*. If a tag prescribing a step becomes available, then the step is *prescribed*. If all items for a step are available, the step becomes *inputs-available*. If a step is both inputs-available and prescribed, then it is *enabled* and may execute. The program terminates when no step is currently executing and no unexecuted step is enabled. A valid program termination occurs when a program terminates and all prescribed steps have been executed.

The CnC model permits many run-time system designs, including those for distributed memory systems using MPI, as well as shared memory versions [1]. We specifically evaluate the Intel Linux CnC (v0.3) [3], which is based on Intel’s Threading Building Blocks.

### 3. Tiled Cholesky in CnC

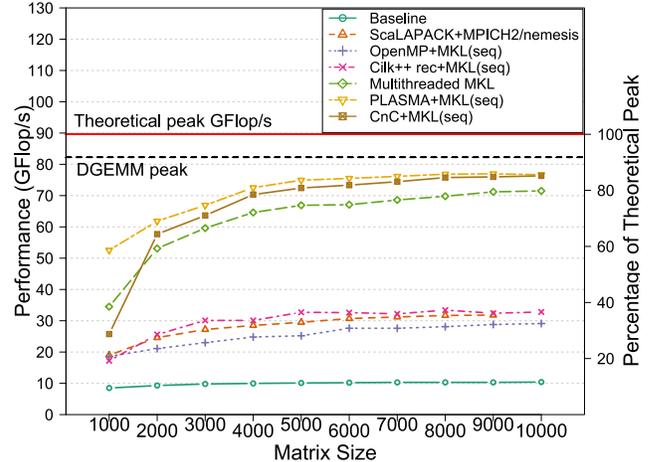
A Cholesky factorization of a symmetric positive definite matrix  $B$  is the product  $L \cdot L^T$ , where  $L$  is a (lower) triangular matrix. We specifically consider the *tiled Cholesky* algorithm of Buttari, *et al.* [2]. This algorithm is based on decomposing  $B$  into blocks (or tiles), and computes  $L$  in an asynchronous-parallel manner suitable for multicore hierarchical memory platforms.

The tiled algorithm consists of three steps: the conventional sequential Cholesky, triangular solve, and the symmetric rank- $k$  update. These steps can be overlapped with one another after initial factorization of a single block, resulting in an asynchronous-parallel approach. There is also abundant data parallelism within each of these steps.

The asynchronous behavior maps naturally to the CnC constructs seen in Section 2. CnC translates the textual representation indicating the graph into C++ code containing stubs for the programmer to fill in. That is, at this point, all the programmer does is input the appropriate tags and data items along with the serial logic of the step. For the serial step implementation, we call tuned sequential vendor BLAS routines. This allows us to couple CnC with an optimized serial library to obtain an efficient parallel asynchronous implementation with minimal programming effort.

We evaluate this CnC implementation on a dual-socket, quad-core 2.8 GHz Intel Xeon X5560 (Nehalem) system. Figure 3 gives the best parallel performance achieved (in GFlop/s) for various matrix sizes. We use a theoretical flop count of  $n^3/3$  when reporting performance.

Our performance baseline is sequential MKL, which runs at about 10 GFlop/s. We also compare with several other implementations of the Cholesky algorithm. The OpenMP-parallel approach is 2.8× faster than the baseline; recursive Cilk++ (v1.0.3) with sequential MKL, and ScaLAPACK (v1.8.0) and MPI (MPICH2 v1.0.8 with the Nemesis device), are an additional 11% faster. Compared to these implementations, our CnC code delivers very high performance, with a speedup of nearly 7.3× on 8 cores. We also achieve more than 85% of the theoretical peak performance for the largest matrix size ( $n = 10,000$ ), and match the performance of the tuned PLASMA (v2.0.0) and Intel MKL (v10.2) implementations.



**Figure 2.** Performance summary of double-precision Cholesky factorization on a dual-socket, quad-core Intel Nehalem system.

### 4. Conclusions and Future Work

The CnC model complements existing approaches for expressing and scheduling asynchronous-parallel computations, providing clean abstractions that enable a variety of control and data flow constructs to be expressed in a way that is both natural and amenable to effective parallelization. We can match or exceed a highly-tuned vendor library for Cholesky factorization, suggesting the promise of CnC as a platform for implementing more asynchronous dense linear algebra kernels, and also for entirely different computation domains. We are currently working on a generalized symmetric dense eigensolver in CnC. Our preliminary results indicate a 2.6× speedup on the Nehalem system when compared to Intel’s MKL implementation, and the performance is within 10% of the theoretically optimal schedule.

### References

- [1] Z. Budimčić, A. Chandramowlishwaran, K. Knobe, G. Lowney, V. Sarkar, and L. Treggiari. Multi-core implementations of the Concurrent Collections programming model. In *Proc. Workshop on Compilers for Parallel Computing (CPC)*, January 2009.
- [2] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. Technical Report UT-CS-07-600 (LAPACK Working Note 191), University of Tennessee Knoxville, September 2007.
- [3] Intel® Concurrent Collections for C++. <http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc/>, 2009.
- [4] K. Knobe. Ease of use with Concurrent Collections (CnC). In *Proc. USENIX Workshop on Hot Topics in Parallelism (HotPar)*, March 2009.
- [5] H. Ltaeif, J. Kurzak, and J. Dongarra. Scheduling two-sided transformations using algorithms-by-tiles on multicore architectures. Technical Report UT-CS-09-637 (LAPACK Working Note 214), University of Tennessee Knoxville, February 2009.
- [6] J. M. Perez, R. M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multicore architectures. In *Proc. IEEE Int’l. Conf. Cluster Computing (CLUSTER)*, pages 142–151, September 2008.
- [7] E. Chan, E. S. Quintana-Ortí, G. Quintana-Ortí, and R. van de Geijn. SuperMatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In *Proc. ACM Symp. Parallelism in Algorithms and Architectures (SPAA)*, pages 116–125, June 2007.