

# Performance Evaluation of Concurrent Collections on High-Performance Multicore Computing Systems

Aparna Chandramowliswaran<sup>†</sup>, Kathleen Knobe<sup>\*</sup>, Richard Vuduc<sup>†</sup>

<sup>†</sup>College of Computing, Georgia Institute of Technology, Atlanta, GA

<sup>\*</sup>Software Solutions Group, Intel Corporation, Hudson, MA

## Abstract

*This paper is the first extensive performance study of a recently proposed parallel programming model, called Concurrent Collections (CnC). In CnC, the programmer expresses her computation in terms of application-specific operations, partially-ordered by semantic scheduling constraints. The CnC model is well-suited to expressing asynchronous-parallel algorithms, so we evaluate CnC using two dense linear algebra algorithms in this style for execution on state-of-the-art multicore systems: (i) a recently proposed asynchronous-parallel Cholesky factorization algorithm, (ii) a novel and non-trivial “higher-level” partly-asynchronous generalized eigensolver for dense symmetric matrices.*

*Given a well-tuned sequential BLAS, our implementations match or exceed competing multithreaded vendor-tuned codes by up to 2.6×. Our evaluation compares with alternative models, including ScaLAPACK with a shared memory MPI, OpenMP, Cilk++, and PLASMA 2.0, on Intel Harpertown, Nehalem, and AMD Barcelona systems. Looking forward, we identify new opportunities to improve the CnC language and run-time scheduling and execution.*

## 1 Introduction and Scope

We study the use of a novel general-purpose parallel programming model, called *Concurrent Collections* (CnC) [4, 12, 13]. In CnC, the programmer expresses her computation in terms of high-level application-specific components, partially-ordered only by minimal semantic scheduling (data- and control-flow) constraints (Section 2). This model encourages the programmer to focus on expressing the computation at a high-level without unnecessary serialization and gives the run-time system flexibility in scheduling operations.

Although there are several papers about various aspects of the CnC model, to date there have been no performance demonstrations or evaluations to assess its viability, particularly for high-performance computing (HPC) applications. This paper is the first such performance study. In particular, we ask whether CnC delivers competitive performance on computations with well-defined performance targets and challenging algorithmic characteristics. For our evaluation, we select dense linear algebra computations written for multicore systems in an *asynchronous-parallel* style, by which we mean bulk-synchronous parallel behavior is replaced by more fine-grained task-level parallelism and localized synchronization [5, 7, 15]. This approach (a) is naturally suited to cores with relatively smaller cache or local-store memories, and (b) reduces the degree of synchronization, whose cost may reasonably be expected to increase with increasing core counts. There are numerous successful demonstrations of this approach for dense linear algebra on current multicore systems [5, 7, 14], meaning there are clear and rigorous performance targets.

**Contributions and findings.** Our central contribution is the first extensive study of the performance potential for HPC applications using the CnC model, based on Intel’s v0.3 Linux CnC implementation for shared memory multicore systems [11]. We discuss which aspects of the CnC language and run-time could be improved. Our study is essential to establishing what the potential is for achieving high performance using CnC.

We express and analyze a prior asynchronous-parallel variant of dense Cholesky factorization when written using CnC. When coupled with a well-tuned BLAS, CnC can closely match or exceed the performance and scalability of the vendor-tuned Intel Math Kernel Library (MKL). Our CnC-based code also compares favorably to PLASMA 2.0, a state-of-the-art domain-specific library-based approach (Section 3). Both MKL and PLASMA use an asynchronous-parallel approach, and so consti-

tute the current state-of-the-art.

For additional comparison, we provide results in alternative programming models, including the “off-the-shelf” solution of ScaLAPACK with a shared memory implementation of MPI (MPICH2+nemesis); OpenMP; and Cilk++. The principal difference between CnC and these models is CnC’s natural support for asynchronous execution.<sup>1</sup> Our findings quantify the gap between asynchronous-parallel and bulk-synchronous execution (Section 4).

Finally, we develop a complete CnC implementation of a novel and partly asynchronous-parallel generalized eigensolver for dense symmetric matrices, of which Cholesky is a small component (Section 3). This non-trivial computation is, as far as we know, the first of its kind. As such, we show that it is feasible to express a complex algorithm within CnC. Our implementation outperforms the Intel MKL equivalent by 1.1–2.6× (Section 4).

**Scope.** Importantly, this study is about the *performance potential* of CnC. Such studies are essential for any new parallel programming model to show value for HPC. Our positive findings show there is potential in CnC as far as performance is concerned.

As an evaluation of CnC for HPC, our use of dense linear algebra limits the findings’ generalizability to one class of computations. Still, this class has challenging properties (e.g., our eigensolver), and so we believe that the basic CnC approach could still be an appropriate starting point for similarly asynchronous-parallel algorithms in other areas.

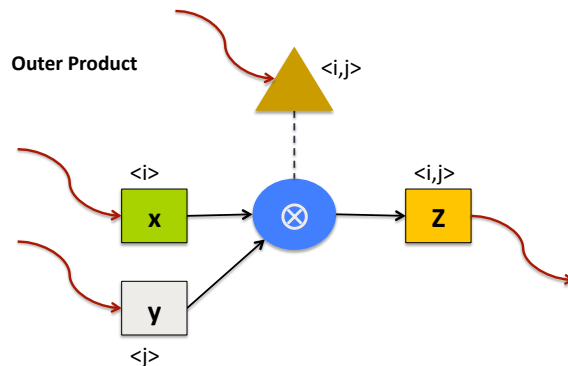
Equally important to questions of performance are those of *productivity*. We argue, qualitatively, that CnC is suitable for these computations. However, we stress that a *true assessment* is a human-factors question, requiring a separate and carefully controlled experiment, and as such is beyond the scope of the present study.

## 2 Overview of CnC

This section provides a cursory overview of the basic CnC concepts relevant to our implementation and experimental results. Portions of this material are taken from existing detailed summaries [4, 11, 12].

The CnC model separates the specification of the computation from the expression of its parallelism. This design can simplify the tasks of a domain expert, who is responsible for expressing the computation, from the tasks of a parallelization and tuning expert (possibly the same person, a different person, or some software/compiler), who identifies the

<sup>1</sup>We do not use the most advanced features of OpenMP 3.0 and Cilk++, nor do we compare directly to the FLAME/SuperMatrix and library-based SMPs approaches [7, 15]. Nevertheless, for Cholesky, we would expect at best comparable performance to PLASMA.



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

parallelism and performs scheduling/distribution and manages communication/synchronization. CnC combines ideas from earlier language work on tuple-spaces, streaming, and data flow models [6, 9, 16] (see Section 5).

**Computation specification.** We summarize the basic CnC model by an example. 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 for the outer product. This graph has 3 kinds of nodes: computational *steps*, data *items*, and control *tags*. Directed edges show producer-consumer relationships among these nodes.

A *step* is the basic unit of execution, which for the outer product is pairwise element multiplication.<sup>2</sup> The blue oval in Figure 1 is a *step collection*, which statically represents the set of dynamic instances of these multiplications.

Data is represented using *item collections*. Here,  $x$ ,  $y$ , and  $Z$  are the three item collections, shown by boxes in Figure 1. Each item collection comprises *item instances*, which in this case are the elements of the  $x$ ,  $y$ , and  $Z$  objects. These items serve as the basic unit of storage, communication, and synchronization.

Steps may consume items (item  $\rightarrow$  step) or produce them (step  $\rightarrow$  item), shown by directed edges in Figure 1.

Each instance of a step or item has a unique application-specific identifier, or *tag*, which is a tuple of *tag components*. For the outer product, it is natural to use element indices as tags. In Figure 1, we denote the

<sup>2</sup>We consider this very fine granularity for example only, as in practice one might wish to choose a larger grain, such as a block.

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/space; 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 connecting the tag collection to the step collection. Multiple step collections may be prescribed by the same tag collection. Importantly, tags indicate *whether* a step will execute, but nothing about *when* it executes. This distinction 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, Figure 1 contains “squiggly” lines that are missing either a source or a sink. These lines mean that 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. (There are other designs; for instance, we could have added an additional step that consumes data containing, say, the dimensions of the  $x$  and  $y$  vectors, and then produces the control tags that prescribe the pairwise multiply step.)

**Textual notation.** There is a formal textual representation of this graph. We illustrate this representation in Section 3, when we describe the CnC implementations of our target dense linear algebra computations. In the current implementation, a translator converts this specification into C++ code, generating subroutine stubs corresponding to the steps. The programmer must implement these stubs (presumably as purely sequential code).

When the run-time calls the sequential step code, it provides the tag and data item instances. The step code calls an API to *get* the input tags and, if it produces tags, *put* them back “into” the graph. We refer the interested reader to the CnC documentation [11].

**Semantics and execution.** If a step executes and produces an item or a tag, that item or tag becomes *available*. If a tag collection prescribes a step collection and a particular tag becomes available, then the step is *prescribed*. If all items for a particular 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 executing and no unexecuted step is enabled. This termination is valid if 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 [4]. We use Intel CnC 0.3, which is based on the Intel Thread Building Blocks (TBB) [11]. The TBB run-time system is based on a Cilk-style work stealing scheduler, with work queues implemented to use last-in first-out (LIFO) order.

In the Intel CnC, there are four types of *events*: start, complete, idle, and requeue. *Start* signals the beginning of execution of a step, while *complete* signals its successful completion. An *idle* event is the time spent between the end of one step and start of the next, when the thread is waiting to be scheduled or waiting for data to become available.

The *requeue* event is specific to the Intel CnC. The run-time may start executing a step as soon as the prescribing tag is available. Thus, if some of the step’s inputs are not yet available, the step may be requeued and tried again later. We revisit requeuing in Section 4.

### 3 Dense Linear Algebra in CnC

---

**Algorithm 1:** Tiled Cholesky factorization algorithm of Buttari, et al. [5].

---

**Input:** Input matrix:  $B$ , Matrix size:  $n \times n$  where  $n = p * b$  for some  $b$  which denotes the tile size

**Output:** Lower triangular matrix:  $L$

```

1 for  $k = 1$  to  $p$  do
2   ConventionalCholesky( $B_{kk}, L_{kk}$ );
3   for  $j = k + 1$  to  $p$  do
4     TriangularSolve( $L_{kk}, B_{jk}, L_{jk}$ );
5     for  $i = k + 1$  to  $j$  do
6       SymmetricRank-
         kUpdate( $L_{jk}, L_{ik}, B_{ij}$ );

```

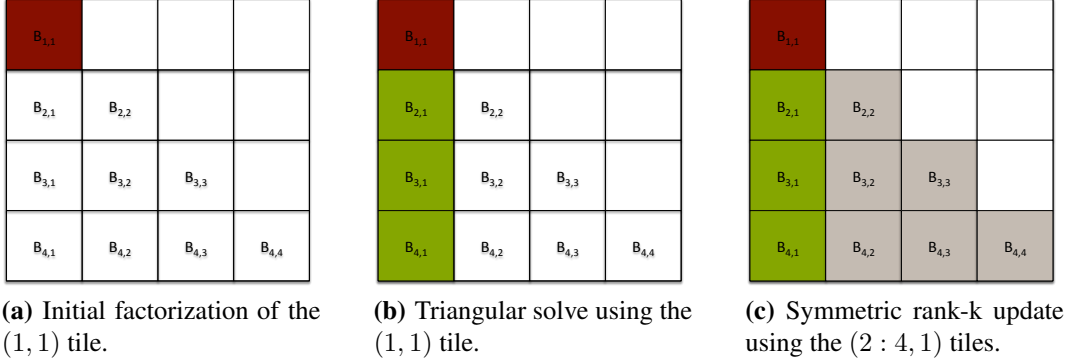
---

In this section, we discuss the CnC implementations of the asynchronous-parallel variant of dense Cholesky factorization and a novel asynchronous-parallel dense generalized eigensolver.

#### 3.1 Asynchronous-Parallel Cholesky

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 Algorithm 1, which is the *tiled Cholesky* algorithm of Buttari, et al. [5]. 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.

Figure 2 illustrates how asynchronous-parallelism



**Figure 2. An illustration of asynchronous Cholesky factorization.**

arises in Algorithm 1. We first factor  $B_{11} = L_{11} \cdot L_{11}^T$  (line 2 of Algorithm 1 for  $k = 1$ ), using a conventional sequential Cholesky algorithm. Then, lines 3–4, which operate on blocks  $B_{21}$ ,  $B_{31}$ , and  $B_{41}$  can execute in parallel. Moreover, lines 5–6 suggest that once we complete operating on the  $B_{21}$  block, we can do a symmetric rank-k update of block  $B_{22}$  in one thread while another thread is still, say, performing the triangular solve on block  $B_{31}$ . Hence, there is a lot of task- and data-level parallelism in Cholesky.

### 3.2 Cholesky in CnC

This asynchronous-parallel behavior maps naturally to the CnC constructs seen in Section 2. Lines 2, 4, and 6 in Algorithm 1 map to *steps* in CnC. The index iteration variables of Algorithm 1 constitute a natural choice for *tags* which helps distinguish between different data items (tiles in this case).

Figure 3 shows the graphical computation specification of tiled Cholesky in CnC. For simplicity, we omit the data items from this graph. Below the graph, we show the textual representation of the graph that the programmer might write. CnC translates the textual representation into C++ code containing stubs for the programmer to fill in code, as illustrated in Figure 4. 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 implementation with minimal coding effort. Figure 4 shows the actual step code with the call to *dtrsm* which performs triangular solve. The API calls (*Get/Put*) before and after the BLAS function call reads in the input tile(s) identified by the tag, performs the computation and outputs tile(s) with the corresponding tag identifier. The input/output and computation performed might vary across different steps, but the basic

principle is the same.

Note that the choice of tags is important, as it determines the amount of parallelism exposed. Tag choice is largely natural for dense linear algebra but a poor choice of tags could impede performance.

Once, the data is available and the step inputting the data is prescribed by a valid tag identifier, the CnC runtime schedules that particular step instance for execution. Given the freedom to schedule steps, CnC schedules them in a way to expose asynchronous-parallel execution.

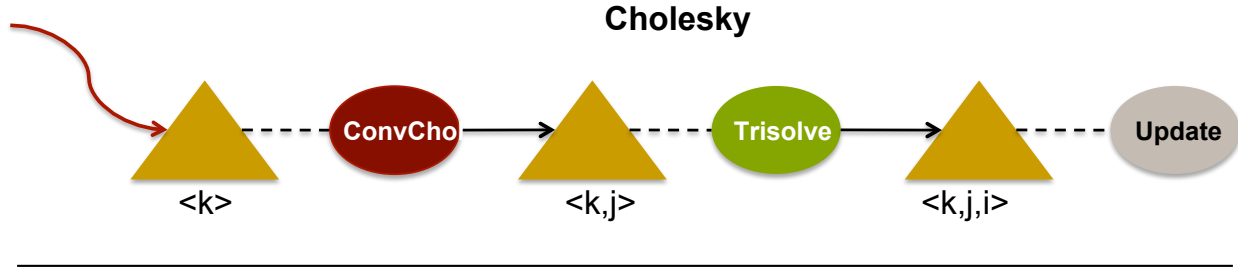
As with related approaches, this CnC-based implementation contains a tuning parameter, the block size, which we have assumed the domain expert introduces and selects. This parameter is critical to performance in that it implicitly controls the degree of available asynchronous-parallelism.

### 3.3 Generalized Symmetric Eigensolver

Though illustrative, the Cholesky example is fairly compact. To better understand and assess CnC, we also developed a tiled and partly asynchronous-parallel symmetric generalized eigensolver, which is considerably larger than Cholesky. This is one of the first efforts on designing and implementing an asynchronous eigensolver and most importantly a model like CnC allows us to express the asynchronous-parallel behavior naturally with relatively modest effort.

**Algorithm.** To compute the eigenvalues,  $\lambda$  we solve the linear algebra equation  $Az = \lambda Bz$ . Here,  $A$  is symmetric and  $B$  is symmetric positive definite matrix. Although this implementation is based on the LAPACK routine *dsygvx*, we note that unlike the original algorithm our implementation in CnC is in fact asynchronous-parallel.

The basic algorithm implemented by *dsygvx* has four components. First, we compute the Cholesky factorization  $B \rightarrow LL^T$ . Next, we reduce the real symmetric-definite generalized eigenproblem to so-called standard



```

// Item: Matrix L, tagged by <k, j, i>
// Suppose 'BlockedMatrix<T>' is a C++ class
// that encapsulates the matrix data,
// dimension 'n' and block size 'b'.
[BlockedMatrix<double>* L: int, int, int];

// Tag declarations
<C_tag : int>; // <k>
<TS_tag: int, int>; // <k, j>
<U_tag : int, int, int>; // <k, j, i>

// Step Prescriptions
<C_tag> :: (ConvChol);
<TS_tag> :: (Trisolve);
<U_tag> :: (Update);

// Input from the environment
env -> [L], <C_tag>;

// Step executions
[L] -> (ConvChol);
(ConvChol) -> [L], <TS_tag>;

[L] -> (Trisolve);
(Trisolve) -> [L], <U_tag>;

[L] -> (Update);
(Update) -> [L];

// Output to environment
[L] -> env;

```

**Figure 3. Top: CnC graphical notation of Cholesky factorization. The red oval is the conventional Cholesky step; the green oval is the triangular solve step; and the grey oval is the symmetric rank-k update. Bottom: Textual notation of Cholesky factorization. Includes one statement for each relation in the graph.**

form,  $(L^{-1} * A * L^{-T})z = \lambda z$ . Methods exist for computing the eigenvalues directly from the generalized form. The third component is reduction of the symmetric matrix to symmetric tridiagonal form, using an orthogonal similarity transformation,  $T = Q' * A * Q$ . This step can be decomposed into a number of kernels, including matrix-vector multiplication, symmetric matrix vector multiplication, and dot product, among others. Finally, we extract the eigenvalues from the tridiagonal matrix using a modified QR algorithm. This step is not compute intensive and may be computed by a single thread.

**Asynchronous-parallelism.** Figure 5 is a directed acyclic graph (DAG) of the first two steps of the eigensolver for a matrix partitioned into a  $3 \times 3$  grid of blocks. Nodes represent computation and edges represent dependencies among them. Each block is labeled by the appropriate submatrix block coordinates. Note that all nodes at any level of the DAG (highlighted by a grey oval) have no dependencies among themselves. Although initially the computation is sequential until root node finishes execution, there is abundant parallelism thereafter. As is evident from the figure, different tasks, denoted by different colors, can execute concurrently.

In the eigensolver, we not only execute steps in Cholesky asynchronously, but also interleave them with steps of the reduction to standard form (e.g., right trian-

gular solve, symmetric matrix multiplication). The CnC code easily expresses this concurrency, and the run-time exploits that concurrency naturally through its scheduling as discussed in Section 4.

Although it is possible to extract parallelism using CnC from the third component of the eigensolver (reduction to tridiagonal form), the inherent dependencies inhibit asynchronous-parallel execution. We would need a different algorithmic approach altogether.

**Parallelization.** Once the dependencies between the steps are laid out, it is possible to extract parallelism more efficiently. To that end, we parallelize all sub-kernels within the first three phases. Unfortunately, one of the most compute-intensive kernel is the symmetric matrix-vector multiply (dsymv), which was not efficiently implemented in the BLAS. Hence, for this kernel alone, we manually parallelize the computation in CnC. This trade-off is worth while as we observe a dramatic increase in performance for a slight increase in programmer effort.

## 4 Results and Discussion

In this section, we first evaluate our CnC-based Cholesky and symmetric generalized eigensolver implementations on the three state-of-the-art multicore platforms shown in Table 1. We then compare their performance to six other implementations (double-precision)

Vendor	AMD	Intel	Intel
Proc. Model	Opteron 8350	Xeon E5405	Xeon X5560
Proc. Name	Barcelona	Harpertown	Nehalem
Clock(GHz)	2	2	2.8
# Sockets	4	2	2
Cores(Threads)/Socket	4(4)	4(4)	4(8)
L1 Data Cache	64 KB/core	32 KB/core	32 KB/core
L2 Data Cache	512 KB/core	6 MB/2cores	256 KB/core
Shared L3 Cache	2 MB/socket	–	8 MB/socket
DRAM Capacity	32 GB	4 GB	12 GB
DRAM Bandwidth (GB/s)	21.3	21.3	51.2
DP Peak Performance (GFlop/s)	128	64	89.6

Table 1. Evaluation platforms for our experiments.

```

StepReturnValue_t Trisolve(
cholesky_graph_t& graph,
const Tag_t& TS_Tag) {

char uplo = 'l', side = 'r';
char transa = 't', diag = 'n';
double alpha = 1;

const int k = (TS_Tag[0]);
const int j = (TS_Tag[1]);

// For each input item in this step
// retrieve the item using the proper tag

// User code to create item tag here
BlockedMatrix<double>* A_block =
graph.L.Get(Tag_t(j, k, k));
BlockedMatrix<double>* Li_block =
graph.L.Get(Tag_t(k, k, k+1));

// Get block size
int b = A_block->getBlockSize ();

// Step implementation logic goes here
dtrsm(&side, &uplo, &transa, &diag, &b, &b,
&alpha, Li_block, &b, A_block, &b);

// For each output item for this step
// put the new item using the proper tag

// User code to create item tag here
graph.L.Put(Tag_t(j, k, k+1), A_block);

return CNC_Success;
}

```

Figure 4. CnC code for the triangular solve step of the Cholesky algorithm. The black and gray text in this code snippet denote the stubs that are generated automatically using the inputs and outputs defined in the graph. The code fragments filled in by the user are indicated in bold (blue color text). Note: We call tuned BLAS for the sequential step implementation (*dtrsm* in this example).

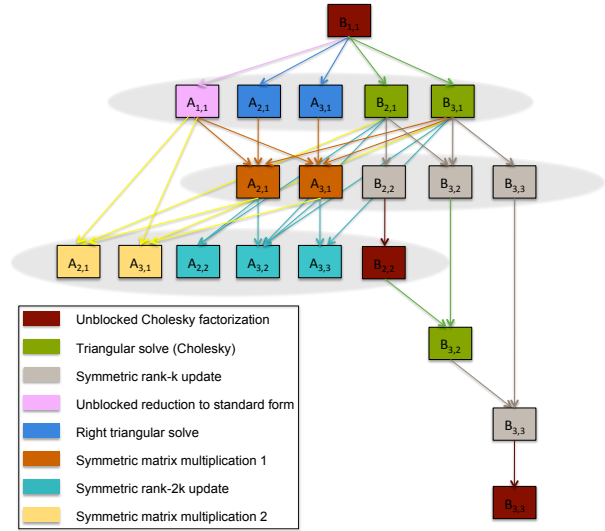


Figure 5. DAG representation of the eigen-solver.

and execution-time bounds based on critical path length. For the non-CnC implementations, we make a “best effort” to do some tuning, and always use sequential MKL when possible. Finally, we examine CnC’s scheduling compared to bulk-synchronous strategies.

We compare the following implementations.

**Baseline – Sequential MKL:** The Intel Math Kernel Library (MKL) implementation of Cholesky factorization, *dpotrf*, run in sequential mode with the input matrix in column-major storage. This baseline is highly tuned by Intel. We also measure multithreaded MKL (see below). [In the plots, we use sequential MKL implementation as the baseline and show this by *hollow circles*.]

**Blocked iterative OpenMP + sequential MKL:** We implemented the tiled Cholesky Algorithm 1, where we

(1) distribute the loop in line 3 to get two ‘j’ loop nests, i.e., one for triangular solve and one for symmetric rank- $k$  update; and then (2) use OpenMP to parallelize the ‘j’ loops. We then use the highly-tuned sequential MKL for each block operation. We tune the block size by exhaustive search for each input size, and report the best performance. [*plus signs*]

**Cilk++ 1.0.3 block recursive + sequential MKL:** We implemented a blocked recursive Cilk++ implementation. In particular, the entire algorithm including the triangular solve and rank- $k$  update steps are performed recursively [2], so as to be able to easily use the Cilk++ thread `spawn` keyword. The recursive form of each step partitions the matrix into roughly half in each dimension. We stop recursion at a tunable block size, determined by exhaustive search for each input matrix dimension. We report the best performance. We use sequential MKL for the leaf kernels. [*crosses*]

**Multithreaded MKL:** We use the multithreaded MKL implementation of Cholesky factorization, `dpotrf`. We report performance on the the number of cores that delivers the highest performance, up to the maximum available cores. The input matrix is in column-major storage. [*hollow diamonds*]

**ScaLAPACK + shared memory MPI:** We use ScaLAPACK 1.8.0 with an MPI “tuned” for shared memory. In particular, we use MPICH2 1.0.8 compiled with the Nemesis device. We tune the processor grid, trying all valid configurations for a given number of MPI tasks, trying all numbers of tasks, and report the best performance. [*up-pointing triangles*]

**PLASMA + sequential MKL:** We use the Cholesky implementation that is part of freely available PLASMA 2.0.0 package. There is a block size parameter, which we tune for each problem size. Since PLASMA currently does not solve eigenvalue problems, we compare only against our Cholesky. [*downward pointing triangles*]

**CnC + sequential MKL:** The CnC implementation of Cholesky using sequential MKL for the steps. The data is stored in blocked data layout [2, 5, 7, 10]. The block size used in the layout is determined by an exhaustive search over all possible values for a given input matrix size. The block size that achieves the highest performance is chosen. [*filled squares*]

**DGEMM Peak:** The peak performance (GFlop/s) of double-precision dense matrix multiplication measured using all the cores on the system. Rather than benchmarking all sizes, we show a representative GFlop/s number for  $n = 10,000$ , which gave the best DGEMM performance on all three platforms among all values of  $n$  that we considered for Cholesky and the eigensolver. [*dashed lines*]

**Theoretical Peak:** The theoretical machine-specific

upper-bound on double-precision GFlop/s achievable. [*solid lines*]

**Compilers:** For our CnC Cholesky factorization, eigensolver, OpenMP and ScaLAPACK implementations, we use the Intel v11.0 and v10.1 compilers on the Intel and AMD platforms respectively. We use MKL v10.0.3.020 on Barcelona, v10.2 on Nehalem, and v10.1.0.019 on Harpertown. For our Cilk++ implementation, we use the gcc 4.2 compiler that ships with it.

## 4.1 Cholesky factorization

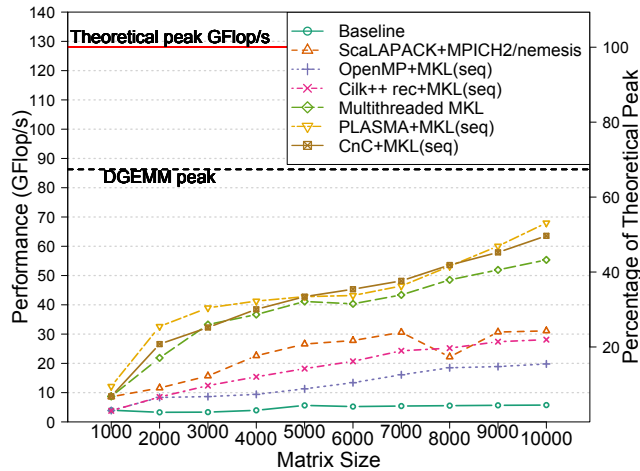
Figure 6 presents the performance scalability by architecture as a function of matrix size for double precision Cholesky factorization. The baseline performance results correspond to sequential MKL values. On all machines, we use the number of cores that delivers highest performance for each matrix. We use the theoretical flop count of  $n^3/3$  when reporting performance.

On Nehalem, our asynchronous-parallel CnC Cholesky compares favorably to PLASMA and MKL. The sequential MKL baseline runs at over 10 GFlop/s. By contrast, OpenMP with sequential MKL is  $2.8\times$  faster than the baseline; and recursive Cilk++ with sequential MKL and ScaLAPACK using shared memory MPI provides only an additional 10% over that. We observe that our fully asynchronous-parallel CnC implementation using sequential MKL delivers very good scalability (speedup of nearly  $7.3\times$  in comparison to the baseline), upto 8 threads. Beyond this, HyperThreading yields no added benefit on all three competing implementations (MKL, PLASMA and CnC). Nevertheless, CnC Cholesky on Nehalem achieves more than 85% of the theoretical peak performance for the largest matrix size ( $n = 10,000$ ) where DGEMM is at 92%.

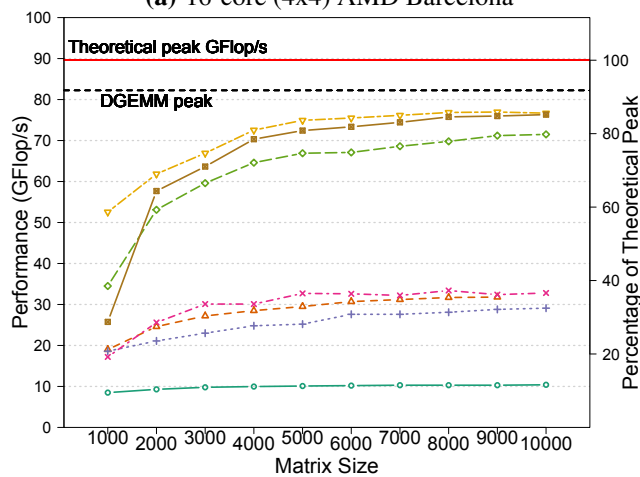
We make similar observations on Harpertown whose Core architecture is similar to Nehalem. Unlike Nehalem, there is no simultaneous multithreading (SMT) on Harpertown. Once again, our CnC implementation achieves near perfect scaling, a speedup of  $7.5\times$  on 8 cores, competing well with MKL and PLASMA implementations. Moreover, we achieve more than 80% of the theoretical peak performance.

The data on Barcelona also follow similar trends except, interestingly, Cholesky factorization achieves only half the theoretical peak performance. Nevertheless, our CnC implementation delivers performance on par with the state-of-the-art PLASMA and exceeds multithreaded MKL for large problem sizes. Barcelona performance also shows good scalability, nearly  $11\times$  on 16 cores. (Note: We did check AMD’s BLAS, which was slower than MKL.)

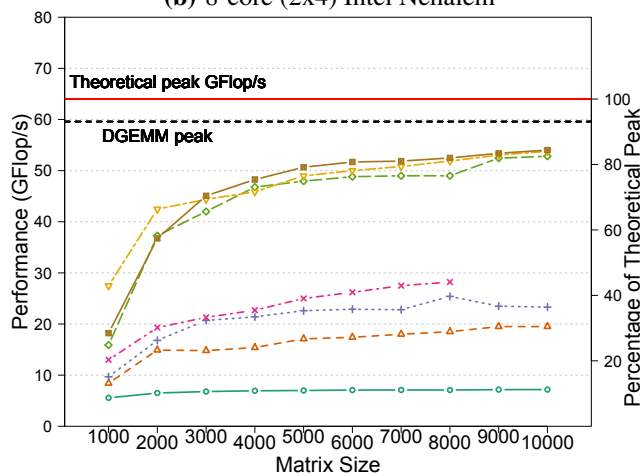
In summary, these results show the potential of CnC to exploit the available parallelism, achieving competi-



(a) 16-core (4x4) AMD Barcelona

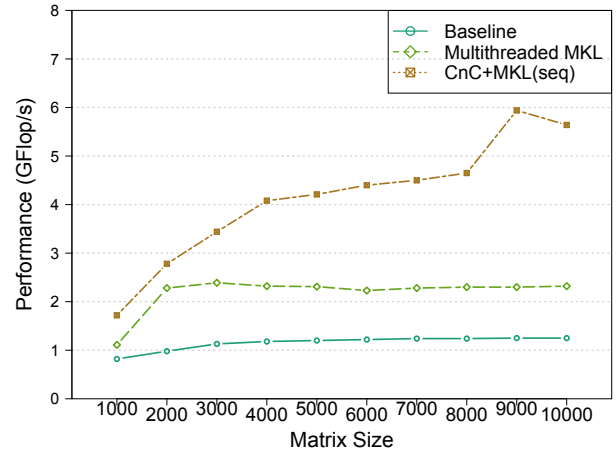


(b) 8-core (2x4) Intel Nehalem

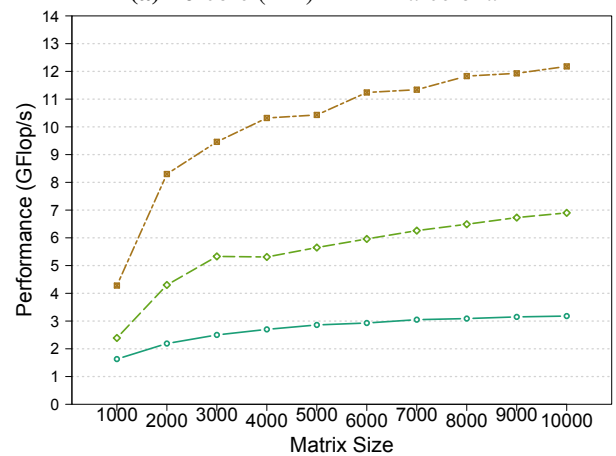


(c) 8-core (2x4) Intel Harpertown

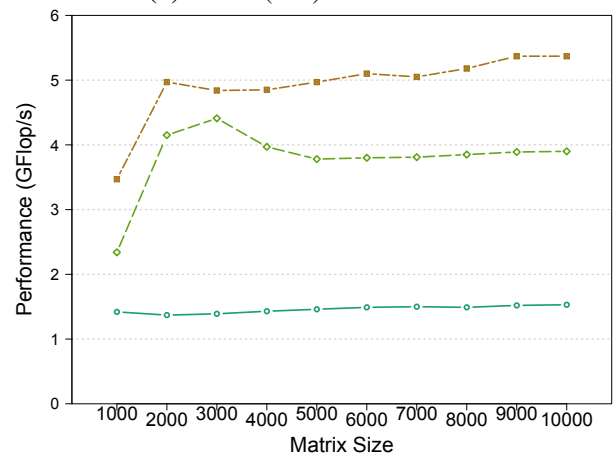
**Figure 6. Performance summary of double precision Cholesky factorization: Performance in GFlop/s (left y-axis) and percentage of theoretical peak (right y-axis) as a function of matrix size, comparing seven implementations discussed.**



(a) 16-core (4x4) AMD Barcelona



(b) 8-core (2x4) Intel Nehalem



(c) 8-core (2x4) Intel Harpertown

**Figure 7. Performance summary of double precision Eigensolver: Performance (GFlop/s) for the three implementations discussed. The flop count used is measured using PAPI performance counters.**



tive performance with reasonable programming effort.<sup>3</sup>

## 4.2 Eigensolver performance

Figure 7 compares the double precision eigensolver performance on our three machines. For all platforms, we compare: (i) the baseline sequential MKL implementation; (ii) the multithreaded MKL implementation; and (iii) our CnC code. Though all three codes can compute both eigenvalues and eigenvectors, we compute just the eigenvalues since it is generally recognized that `dgsyvx` is best suited to that case.

We observe that CnC delivers significantly higher performance than multithreaded MKL on all three systems, with speedups of 1.9 $\times$ , 2.6 $\times$ , and 1.5 $\times$  in the best case for Nehalem, Barcelona, and Harpertown, respectively.

Three factors contribute to this improvement. First, the critical path of the asynchronous-parallel CnC implementation is smaller than multithreaded MKL due to a reduced number of synchronizations. Secondly, the symmetric matrix-vector multiply kernel is not parallelized in MKL, as we confirmed by testing. Finally, on Barcelona and Nehalem, NUMA effects likely play an additional role as well, causing the MKL eigensolver implementation to not scale beyond 1 socket. Hence we compare against the 4-thread and 8-thread runs only on Barcelona and Nehalem respectively, which was the best result we could obtain when searching over all numbers of threads up to 16 (to match the 16 cores on Barcelona and 8 cores hyperthreaded on Nehalem).

Unlike MKL, in our CnC implementation we manually parallelized the symmetric matrix-vector multiply routine, thereby enabling the computation to scale up to the maximum number of cores/threads.

Interestingly, we do not see scalability issues on Harpertown. The MKL eigensolver scales up to 8 threads even though the symmetric matrix-vector multiply is sequential. Our partly asynchronous-parallel CnC implementation is up to 1.5 $\times$  faster than multithreaded MKL. Even though the eigensolver implementation is non-trivial and much more complex than Cholesky factorization, our CnC implementation achieved a high level of performance, showing that at least the basic model and run-time system have good potential. We refer interested readers to [8] which contains additional details on comparison of CnC against other asynchronous approaches, detailed scalability results, which we omit in this paper due to space constraints.

---

<sup>3</sup>Although we do not use the most recent version of MKL on Barcelona and Harpertown, we believe the comparisons made are fair in that we use the same MKL for all implementations on a given platform.

## 4.3 Scheduling

The current run-time system is characterized by a static grain size, dynamic schedule, and dynamic distribution. It is built on top of the TBB [1]. TBB controls the scheduling and execution of steps in a CnC program. TBB implements a Cilk-like work-stealing scheduler that supports fine-grained task parallelism [3].

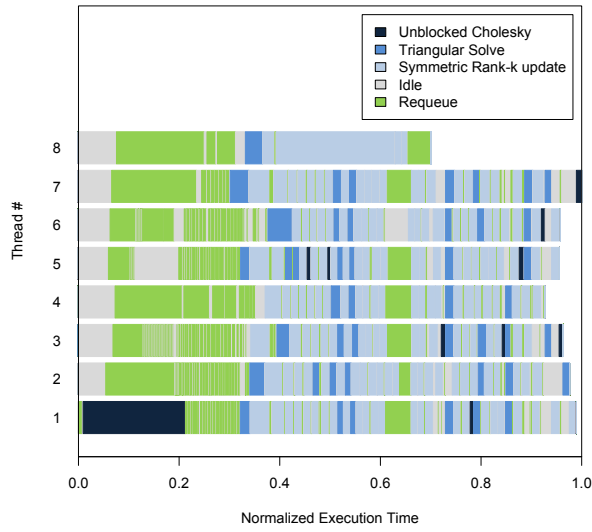
**Tag generation.** In CnC, we have the option either to pre-generate tags or generate them on-the-fly. Figure 8 depicts an execution timeline for the Cholesky factorization of a matrix of size 1000, for two approaches to tag generation. Figure 8(a) shows the approach in which we pre-generate all tags. Owing to the run-time’s LIFO queuing (Section 2), the last tag generated will be scheduled first. That is, for Cholesky, first-in-first-out (FIFO) is preferable. We typically want the first tag value in tag space  $< k >$  to be scheduled first since all other steps are stalled until this step finishes execution. One solution is to generate tags with data on-the fly shown in Figure 8(b).

We layout the execution profile of each thread along the y-axis (one “row” per thread); and on the x-axis show execution time, normalized in both charts to the time taken by longest executing thread. The different color-coded regions represent the different step instances.

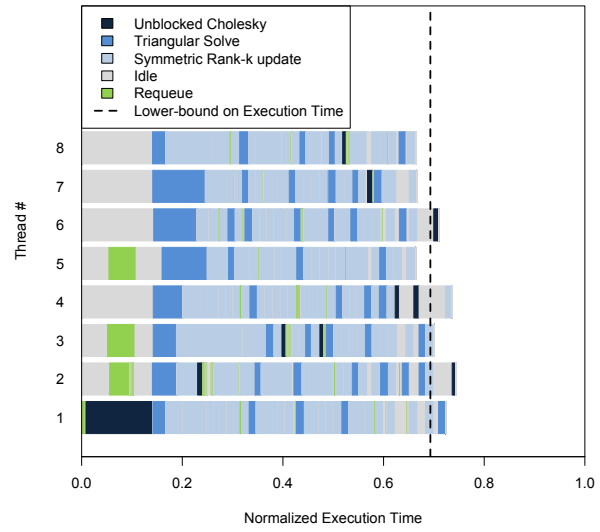
We make a number of observations. First, the dynamic tag approach takes only 75% of the time taken by the pre-generated tags approach. When pre-generating tags, 30.6% of the overall execution time is spent on requeue events for the matrix size 1000 (sum of green regions). The computation is also not load balanced, with thread 8 completing much earlier than the other threads. When we explicitly generate tags only when data becomes available, we observe a marked decrease in the number of requeue events, thereby yielding the 25% improvement in time. Moreover, the computation is better load balanced.

By reducing requeue events, we increase the number of *Get* and *Put* operations, but the overhead due to these operations is much less than the requeue delays, as shown by the decrease in the overall running time.

However, we also observed for larger  $n$ , requeue delays were actually not that significant. In particular, recall from Figure 8 that the time spent on the factorization of the first block  $B_{11}$  is about 20% of the entire execution time. Hence, all other threads waiting for the input from the first step are requeued. When  $n = 6000$ , less than 1% of the time was spent on factoring the first block, and so the time spent on requeue events was only 0.56% of the overall execution time. Thus, the on-the-fly approach does not pay-off in all instances and, in fact, becomes a “tunable” parameter.

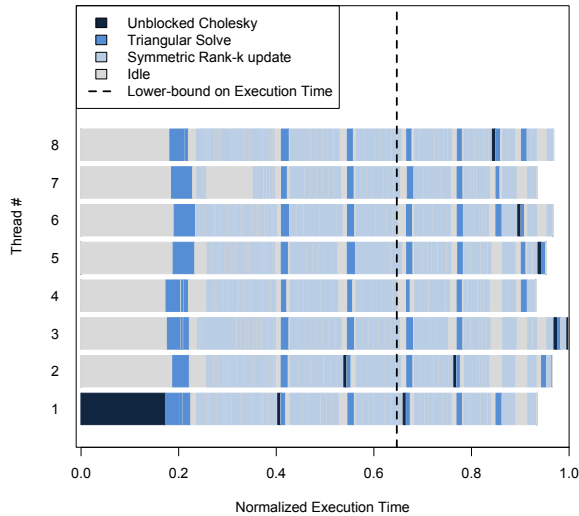


(a) Tags pre-generated; requeue time = 30.6%.



(b) Tags generated when data becomes available.

**Figure 8. Scheduling for Cholesky factorization (matrix size = 1000)**



**Figure 9. Scheduling timeline for Cholesky factorization using Cilk++.**

**Comparison to bulk-synchronous approaches.** It is well-established that asynchronous scheduling can eliminate the idle times present in bulk-synchronous approaches. For example, Figure 9 shows the scheduling of Cholesky factorization in our Cilk++-based recursive

implementation. There is a synchronization stage at the end of every task. Thus, the idle time increases because fast threads are waiting for the slower ones to complete execution. This behavior is a consequence of our choice of a recursive implementation, which is encouraged by the Cilk++ model, as well as the model’s nested DAG parallelism.

In the CnC implementation, there is no synchronization of tasks and the execution driven by tags imposes only one condition on preserving the data dependency between the steps. This eliminates idle time, as is evident when comparing Figures 8(b) and 9. Also, these figures show a vertical dotted line which is the estimated lower-bound on execution time. Given a weighted DAG (node weights measured as the time spent in the corresponding step, and the edge weights to be 0), the lower bound is computed by finding the longest path from the start to end (sequential steps along the critical path). We observe that CnC performs extremely well, within 10% of its lower bound; by contrast, the bulk-synchronous code performs well-below its potential.

Figure 10 shows scheduling of the eigensolver on Harpertown. The scheduling figure only shows the asynchronous-parallel portion of the eigensolver, which has the Cholesky and reduction to standard form components. Figure 10(a) shows how the scheduling unfolds when all the tags have been generated before the start of computation. In this part of the computation, more than 80% of the execution time is spent on requeue events.

This behavior is due to the LIFO scheduling, where tags are scheduled last-in-first-out. Since the number of tag spaces and tags in each tag space are much larger compared to Cholesky factorization, the amount of requeue is significantly higher at start. However, the number of requeue events for the entire computation is only 33.3% of the execution time. Figure 10(b) shows an 85% reduction in overall execution time of the zoomed portion by generating tags only when input becomes available.

In short, we can achieve high performance in CnC, but there is still scope for additional improvements and tuning in the run-time system with respect to scheduling of steps, locality, and data movement.

## 5 Related Work

Existing work on asynchronous-parallel algorithms for dense linear algebra covers Cholesky, LU, and QR factorization, as well as so-called “two-sided” transformations, Hessenberg, tridiagonal and bidiagonal reduction [5, 7, 14]. The implementations are based on some combination of schedulers and APIs based on domain-specific abstraction (e.g., SuperMatrix [7]), or hand-coded or pragma-directed schemes (e.g., SMPs [14, 15]). The present study contributes experience working in a novel model and a novel asynchronous-parallel implementation of a different algorithm (the eigensolver).

The CnC programming model itself has rich influences from the long history of concurrent programming models, including tuple-spaces, streaming languages, and dataflow languages [6, 9, 16]. CnC’s key distinction is its treatment of both control and dataflow, thereby making CnC more general than pure streaming or classical data flow approaches. Also, item collections in CnC allows for more general indexing than dataflow arrays. While both CnC and tuple space languages like Linda specify computation using tags, they differ in a number of aspects. In CnC there is a clear separation between tags and values, while there is no distinction between the two in Linda. Moreover items are accessed by value and not by location, and adhere to dynamic single assignment form, as noted by Budimlic, et al. [4].

## 6 Conclusions and Future Work

This study constitutes the first performance evaluation of the CnC model, with compelling results on a challenging pair of computations from parallel dense linear algebra. CnC complements existing approaches for expressing and scheduling asynchronous-parallel computations, by providing novel abstractions that enable a variety of control flow and dataflow constructs to be expressed in a way that enables effective parallelization. For our target computations, we can both (a) match or exceed a highly-tuned vendor library for Cholesky and (b) extend these results to significant speedups (1.1–

2.6×) on a complicated eigensolver. Indeed, the CnC model *enabled* our novel asynchronous-parallel eigensolver implementation.

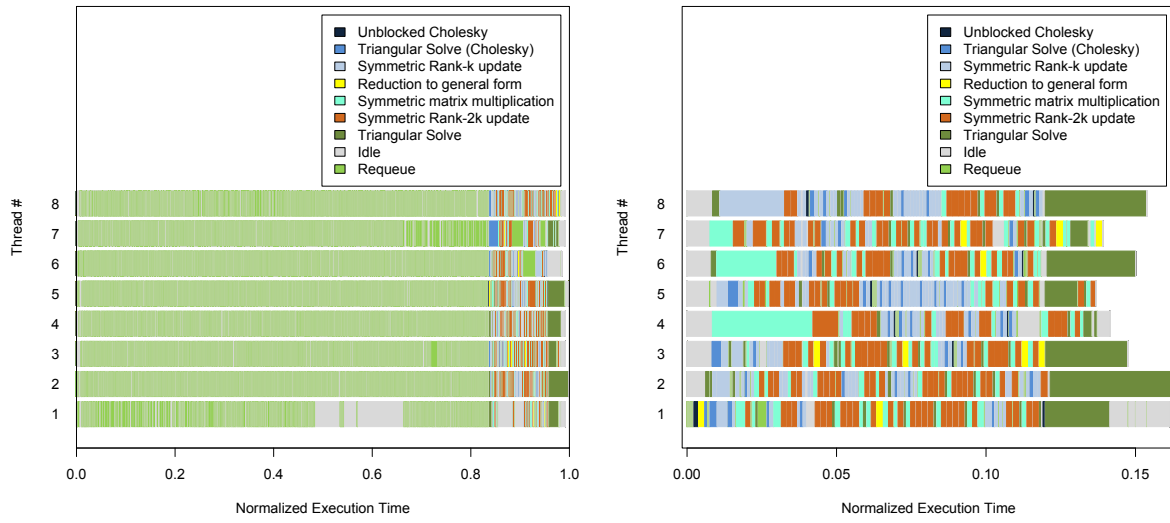
Our experience reveals ways in which to improve CnC further. First, additional work queue scheduling policies (besides LIFO) are needed. Secondly, we can avoid run-time inefficiencies by exploiting additional dependence information available in the specification itself (textual notation). In our case, when the same tag collection prescribes multiple dependent step collections, we can reduce requeuing by not scheduling those collections. Thirdly, there are a number of ways in which tag management could be tuned, perhaps automatically. Finally, there are ways to enhance the textual notation and API; we are currently looking at adding new abstractions as well as new syntax for easily composing CnC components.

## Acknowledgments

This work was supported in part by the National Science Foundation (NSF) under award number 0833136, and grants from the Defense Advanced Research Projects Agency (DARPA) and Intel Corporation. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of NSF, DARPA, or Intel. We also wish to thank Geoff Lowney and Frank Schlimbach of Intel Corporation for helping us collect runtime traces for the scheduling timelines.

## References

- [1] Intel® Threading Building Blocks, 2009. [www.threadingbuildingblocks.org](http://www.threadingbuildingblocks.org).
- [2] Bjarne S. Andersen, Fred Gustavson, Alexander Karaivanov, Minka Marinova, Jerzy Wasniewski, and Plamen Yalamov. LAWRA: Linear algebra with recursive algorithms. In *Appl. Par. Comput.*, volume LNCS 1947, pages 38–51. Springer, 2001.
- [3] Robert D. Blumofe, Christopher F. Jörg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proc. PPOPP*, pages 207–216, 1995.
- [4] Zoran Budimlic, Aparna Chandramowlishwaran, Kathleen Knobe, Geoff Lowney, Vivek Sarkar, and Leo Treggiari. Multi-core implementations of the Concurrent Collections programming model. In *Proc. Wkshp. Compilers for Par. Comput. (CPC)*, 2009.
- [5] Alfredo Buttari, Julien Langou, Jakob Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. Technical Report UT-CS-07-600, Univ. of Tenn. Knoxville, Sep. 2007.
- [6] Nicholas Carriero and David Gelernter. Linda in context. *Comm. ACM*, 32(4):444–458, 1989.
- [7] Ernie Chan, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, and Robert van de Geijn. SuperMatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In *Proc. SPAA*, pages 116–125, 2007.
- [8] Aparna Chandramowlishwaran, Kathleen Knobe, and Richard Vuduc. Performance evaluation of Concurrent Collections on high-performance multicore computing systems. Technical Re-



(a) Pre-generated tags result in 33.3% of time spent on requeue events.

(b) Tags generated on-the-fly.

**Figure 10. Scheduling for Eigensolver (matrix size = 1000)**

port GT-CSE-10-01, Georgia Institute of Technology, Atlanta, GA, USA, February 2010.

[9] Jack B. Dennis. First version of a data flow procedure language. In *Programming Symp.: Proc.*, volume LNCS 19, pages 362–376. Springer, 1974.

[10] Fred G. Gustavson. New generalized data structures for matrices lead to a variety of high performance dense linear algebra algorithms. In *Appl. Par. Comput.*, volume LNCS 3732, pages 11–20, 2006.

[11] Intel® Concurrent Collections for C/C++: User’s Guide, v0.3, 2009.

[12] Kathleen Knobe. Ease of use with Concurrent Collections (CnC). In *Proc. USENIX HotPar*, 2009.

[13] Kathleen Knobe and Carl D. Offner. TStreams: A model of parallel computation. Technical Report HPL-2004-78R1, HP Labs, 2004.

[14] Hatem Ltaeif, Jakub Kurzak, and Jack Dongarra. Scheduling two-sided transformations using algorithms-by-tiles on multi-core architectures. Technical Report UT-CS-09-637, Univ. of Tenn. Knoxville, 2009.

[15] Josep M. Perez, Rosa M. Badia, and Jesus Labarta. A dependency-aware task-based programming environment for multicore architectures. In *Proc. IEEE CLUSTER*, pages 142–151, 2008.

[16] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A language for streaming applications. In *Proc. Int’l. Conf. Compiler Construction (CC)*, volume LNCS 2304, pages 49–84. Springer, 2002.