CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2012; **24**:2282–2301
Published online 26 January 2012 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.2811

Measuring the overhead of Intel C++ Concurrent Collections over Threading Building Blocks for Gauss–Jordan elimination

Peiyi Tang*,†

Department of Computer Science, University of Arkansas at Little Rock, Little Rock, AR 72204, USA

SUMMARY

The most efficient way to parallelize computation is to build and evaluate the task graph constrained only by the data dependencies between the tasks. Both Intel's C++ Concurrent Collections (CnC) and Threading Building Blocks (TBB) libraries allow such task-based parallel programming. CnC also adapts the macro data flow model by providing only single-assignment data objects in its global data space. Although CnC makes parallel programming easier, by specifying data flow dependencies only through single-assignment data objects, its macro data flow model incurs overhead. Intel's C++ CnC library is implemented on top of its C++ TBB library. We can measure the overhead of CnC by comparing its performance with that of TBB. In this paper, we analyze all three types of data dependencies in the tiled in-place Gauss–Jordan elimination algorithm for the first time. We implement the task-based parallel tiled Gauss–Jordan algorithm in TBB using the data dependencies analyzed and compare its performance with that of the CnC implementation. We find that the overhead of CnC over TBB is only 12%–15% of the TBB time, and CnC can deliver as much as 87%–89% of the TBB performance for Gauss–Jordan elimination, using the optimal tile size. Copyright © 2012 John Wiley & Sons, Ltd.

Received 23 August 2011; Revised 11 December 2011; Accepted 11 December 2011

KEY WORDS: task-based parallel execution; graph-driven asynchronous execution; Intel C++ CnC; Intel C++ TBB; data dependency analysis; tiled Gauss–Jordan elimination

1. INTRODUCTION

Coarse-grain loop-based parallel software packages such as LAPACK [1] have shown limitations on emerging multicore architectures [2] because of the large granularity of computation and the over specification inherited in the loop-based fork-join paradigm. The most efficient approach of parallel execution is to build and evaluate the task graph constrained only by the data dependencies between the tasks without over specification [3]. Fine-grain tasks can also fit the data accessed into the small caches of the cores.

Both Intel's C++ Concurrent Collections (CnC) [4] and Threading Building Blocks (TBB) [5, 6] libraries allow such task-based parallel programming. CnC also adapts the macro data flow model [7] by allowing only single-assignment data objects in its Linda-tuple-space-like [8] global space called CnC context. Each computational task in CnC is purely functional and side-effect free. Therefore, only flow data dependencies between the tasks need to be honored, and they can be enforced through the single-assignment data objects in the CnC context. This makes parallel programming in CnC much easier [9, 10].

Intel's C++ CnC library for shared-memory multicore processors is implemented on top of Intel's TBB library [5, 6] for task-based asynchronous multicore parallel programming. TBB is not functional, and tasks in TBB read and write the shared memory of multicore computers directly.

*Correspondence to: Peiyi Tang, Department of Computer Science, University of Arkansas at Little Rock, Little Rock, AR 72204, USA.
†E-mail: pxtang@ualr.edu

Implementing the data flow model on shared memory incurs overhead because the data objects needed by tasks need to be copied from the context. The data produced by tasks also need to be copied to the context. The associative search with tags for data and task objects in the context also incurs overheads.

In principle, every task-based asynchronous parallel computation (also known as graph-driven asynchronous execution) can be specified and executed in both CnC and TBB. The trade-off between the ease of programming and the efficiency of the program has always been a major factor in determining whether or not to adapt a new programming model or language. We are interested in quantifying the overhead of CnC over TBB. In this paper, we use the tiled Gauss–Jordan elimination algorithm for inverting matrices to measure the overhead of CnC over TBB.

The Gauss–Jordan elimination algorithm is one of the several dense linear algebra algorithms used in many scientific applications especially those involving optimization and visualization. It has $O(n^3)$ time complexity using $O(n^2)$ data space. The tiled Gauss–Jordan algorithm uses a block array layout [11] for better cache performance, and the computations on array tiles (blocks) fit the task-based parallel model of CnC and TBB well. The past parallel tiled Gauss–Jordan algorithms either explore only the loop-based parallelism [12] or are restricted on the augmented system using two data arrays [13].

In this paper, we analyze and formulate all the data dependencies between the tasks in the tiled in-place Gauss–Jordan algorithm for the first time and parallelize and implement the algorithm in TBB.

We compare the TBB performance with the performance of the CnC implementation [4]. We find that the overhead of CnC over TBB on Gauss–Jordan elimination is only 12%–15% of the TBB time for the CnC with tuner and 18%–22% of the TBB time for the CnC without tuner, using the optimal tile size. Given that giga floating-point operations per second (GFLOPS) performance is proportional to the reciprocal of execution time, CnC with and without tuner can thus deliver as much as 87%–89% and 82%–85% of the TBB GFLOPS performance, respectively, using the optimal tile size.

The rest of the paper is organized as follows. Section 2 presents the data dependency analysis of all three kinds of dependencies (flow, anti, and output) of the tiled in-place Gauss–Jordan algorithm. Section 3 introduces the CnC model and an implementation of the parallel tiled Gauss–Jordan algorithm in CnC. Section 4 describes the implementation of the task-based parallel tiled Gauss–Jordan algorithm in TBB using the data dependencies analyzed and formulated in Section 2. Section 5 presents our experimental results of both CnC and TBB parallel codes, as well as the sequential tiled code, analyzes their performances, and finds the overhead of CnC over TBB. Section 6 describes work related to ours and future work. Section 7 concludes the paper with a discussion.

2. PARALLELIZING TILED IN-PLACE GAUSS–JORDAN ELIMINATION

Gauss–Jordan elimination is an algorithm to invert a matrix A to A^{-1}. The sequential tiled Gauss–Jordan elimination parallelized in [12,13] inverts A with an identity matrix B and has A^{-1} stored in B when the algorithm is finished. A modified in-place algorithm that inverts A in place and stores A^{-1} in A is shown in Figure 1. The in-place algorithm reduces the number of matrix data arrays used from two to one and the number of task types from six to four. In Figure 1, an $n \times n$ matrix data array is tiled to a $d \times d$ tile array. Each tile is a $t \times t$ data array with $t = n/d$ denoted as A_{ij} $(0 \leqslant i, j \leqslant d-1)$. The k-th $(k = -1, 0, \cdots, d-1)$ version of the tile A_{ij} is denoted as $A_{ij}^{(k)}$ in the algorithm of Figure 1. $A_{ij}^{(k)}$ can also be regarded as the k-th value stored in A_{ij}. The initial value of A_{ij} can be seen as $A_{ij}^{(-1)}$. The initial matrix is the collection of $A_{ij}^{(-1)}$ $(0 \leqslant i, j \leqslant d-1)$, and the final inverted matrix is the collection of $A_{ij}^{(d-1)}$ $(0 \leqslant i, j \leqslant d-1)$.

As can be seen from Figure 1, in each iteration of the outermost loop k, each tile A_{ij} $(0 \leqslant i, j \leqslant d-1)$ is updated once, by one of the four types of computation as shown in Table I. Thus, the computation of updating tile A_{ij} in the k-th $(k = 0, \cdots, d-1)$ iteration of the outermost loop can be regarded as task (k, i, j). The tasks with the same index k are called *the tasks of iteration k*.

```
for $k = 0, d-1$
    $A_{kk}^{(k)} = (A_{kk}^{(k-1)})^{-1}$                          //task type I
    for $j = 0, d-1 \wedge j \neq k$
        $A_{kj}^{(k)} = A_{kk}^{(k)} A_{kj}^{(k-1)}$                    //task type $\mathrm{II}_j$
        for $i = 0, d-1 \wedge i \neq k$
            $A_{ij}^{(k)} = A_{ij}^{(k-1)} - A_{ik}^{(k-1)} A_{kj}^{(k)}$    //task type III
        endfor
    endfor
    for $i = 0, d-1 \wedge i \neq k$
        $A_{ik}^{(k)} = -A_{ik}^{(k-1)} A_{kk}^{(k)}$                   //task type $\mathrm{II}_i$
    endfor
endfor
```

Figure 1. Tiled in-place Gauss–Jordan elimination algorithm.

There are total of d^3 tasks with indexes (k, i, j) $(0 \leqslant k, i, j \leqslant d-1)$. The quantity column of Table I shows the total number of tasks of the type. Table I also shows the number of floating-point operations in each type of task.[‡] Recall that $d = n/t$. The total number of floating-point operations as a function of problem size n and tile size t, denoted as flop(n,t), is as follows

$$\text{flop}(n,t) = \left(\frac{1}{t} + 2\right) n^3 - 2n^2 - (t-1)n. \tag{1}$$

It can also be expressed as a function of problem size n and tile array dimension d, denoted as flop(n,d), as follows

$$\text{flop}(n,d) = 2n^3 + \left(\left(d - \frac{1}{d}\right) - 2\right) n^2 + n. \tag{2}$$

The partial derivative of flop(n,d) with respect to d is $\frac{\partial \text{flop}(n,d)}{\partial d} = n^2(1 + \frac{1}{d^2}) > 0$. Thus, flop$(n,d)$ is an increasing function of d and thus, flop(n,t) is a decreasing function of t. When t changes from 1 to n, flop(n,t) decreases from $3n^3 - 2n^2$ to $2n^3 - 2n^2 + n$.

The most efficient parallel execution of dense linear algebra algorithms on multicore processors is to build the task graph of the data dependencies among the tasks and execute the tasks of the graph by honoring these dependencies [14, 15]. There are three kinds of data dependencies: flow (RAW, read after write), anti (WAR, write after read), and output (WAW, write after write) data dependencies. Task B is flow/anti dependent on task A, denoted as $A \rightarrow B/A \mapsto B$, if it reads/writes the data written/read by task A earlier. Task B is output-dependent on task A if it writes the data written by task A earlier. Flow data dependencies are the essential ones and must be honored in any parallel programming model. Anti and output data dependencies can be eliminated by renaming data objects as in the single-assignment programming models, such as CnC or functional programming

Table I. Type, computation, and number of flops of task (k, i, j).

Type	Condition	Quantity	Computation	No. of flops
I	$i = k \wedge j = k$	d	$A_{kk}^{(k)} = (A_{kk}^{(k-1)})^{-1}$	$2t^3 - 2t^2 + t$
II_j	$i = k \wedge j \neq k$	$d(d-1)$	$A_{kj}^{(k)} = A_{kk}^{(k)} A_{kj}^{(k-1)}$	$2t^3$
II_i	$i \neq k \wedge j = k$	$d(d-1)$	$A_{ik}^{(k)} = -A_{ik}^{(k-1)} A_{kk}^{(k)}$	$2t^3$
III	$i \neq k \wedge j \neq k$	$d(d-1)^2$	$A_{ij}^{(k)} = A_{ij}^{(k-1)} - A_{ik}^{(k-1)} A_{kj}^{(k)}$	$2t^3 + t^2$

[‡]The floating-point operations include addition, subtraction, multiplication, and division between two floating-point numbers.

languages. For the shared memory models such as TBB, the task graph has to include all the flow, anti, and output data dependencies.

The loop-based approach to parallelize the algorithm in Figure 1 would keep the outermost loop k serial and find parallelism in its loop body based on the data dependency analysis between the task types of the same iteration k. Figure 2 shows the flow and anti dependencies between different types of tasks of the same iteration k. Because type II_j and type II_i tasks read tile A_{kk} written by the type I task, type II_j and II_i tasks are flow dependent on the type I task. As type III tasks read tiles A_{kj}, written by the corresponding (with the same j index) type II_j tasks, type III tasks are flow dependent on type II_j tasks (with the same j index). These flow data dependencies are shown by the solid arrows in Figure 2. Because type II_i tasks write the tiles A_{ik} that are read by the corresponding type III tasks (with the same index i) earlier, there is an anti dependency from the type III tasks to the corresponding type II_i tasks, shown by the dashed arrow in Figure 2. There are no output dependencies between the tasks of the same iteration k because every tile is written only once by one of these tasks.

Note that type II_i tasks do not read the tiles A_{ij} written by type III tasks, and they are the only tasks in the current iteration k that execute after type III tasks. Therefore, there are no tasks in the current iteration k that read the data written by type III tasks. There are also no tasks in the current iteration k that read the data written by type II_i tasks because type II_i tasks are the last ones in the iteration.

As long as the flow and anti data dependencies between the task types shown in Figure 2 are honored, the tasks of the same type can be executed in parallel because there are no data dependencies among them. The loop-based parallel Gauss–Jordan elimination, based on the dependency graph in Figure 2, would be as shown in Figure 3. The parallel execution of the tasks of the same type is specified by the parallel `forall` loops[§] in Figure 3, and the flow and anti data dependencies between task types are honored by the sequential order of these parallel loops. Note that a parallel loop cannot start its iterations unless all the iterations of the preceding parallel loop are completed. It is this fork-join synchronization of the parallel loop semantics that enforces all the data dependencies in Figure 2.

However, as shown by previous studies [14–16], the performance of loop-based algorithms is always inferior to that of task-based algorithms, partly because of the over specification of this fork-join synchronization. Another reason is that loop-based algorithms cannot explore the parallelism

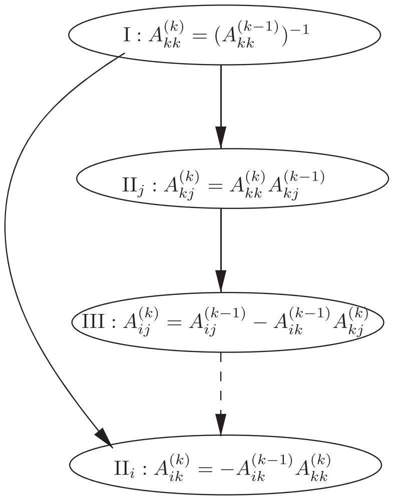

$\mathrm{I} : A_{kk}^{(k)} = (A_{kk}^{(k-1)})^{-1}$
$\mathrm{II}_j : A_{kj}^{(k)} = A_{kk}^{(k)} A_{kj}^{(k-1)}$
$\mathrm{III} : A_{ij}^{(k)} = A_{ij}^{(k-1)} - A_{ik}^{(k-1)} A_{kj}^{(k)}$
$\mathrm{II}_i : A_{ik}^{(k)} = -A_{ik}^{(k-1)} A_{kk}^{(k)}$

Figure 2. Data dependency graph of task types in the same iteration k.

[§]The nested parallel `forall` loops are two-dimensional parallel loop, and all the tasks of type III enclosed can be executed in parallel.

```
for $k = 0, d-1$
    $A_{kk}^{(k)} = (A_{kk}^{(k-1)})^{-1}$                          //task type I
    forall $j = 0, d-1 \wedge j \neq k$
        $A_{kj}^{(k)} = A_{kk}^{(k)} A_{kj}^{(k-1)}$                    //task type II$_j$
    endforall
    forall $j = 0, d-1 \wedge j \neq k$
        forall $i = 0, d-1 \wedge i \neq k$
            $A_{ij}^{(k)} = A_{ij}^{(k-1)} - A_{ik}^{(k-1)} A_{kj}^{(k)}$   //task type III
        endforall
    endforall
    forall $i = 0, d-1 \wedge i \neq k$
        $A_{ik}^{(k)} = -A_{ik}^{(k-1)} A_{kk}^{(k)}$                   //task type II$_i$
    endforall
endfor
```

Figure 3. Loop-based parallel Gauss–Jordan elimination algorithm.

between the tasks in different iterations of the outermost loop k, as it cannot be parallel because of its cross-iteration dependencies.

The dependency graph in Figure 2 cannot be used for task-based parallelization. What we need for task-based parallel execution is a dependency graph between all the individual tasks in the entire algorithm. In the following, we analyze and formulate the flow, anti, and output data dependencies among all the tasks.

2.1. Flow data dependency

Because every tile A_{ij} $(0 \leqslant i, j \leqslant d-1)$ is updated (written) in every iteration of loop k, the tasks that a task in iteration k is flow dependent on are either in the same iteration k or the previous iteration $k-1$. In particular, task (k, i, j) of all types always reads tile A_{ij}, which is written by task $(k-1, i, j)$ (if $k > 0$), and thus, task (k, i, j) is flow dependent on task $(k-1, i, j)$. This is summarized as Lemma flow-0 in Table II, which says that if $k > 0$, then $\{(k-1, i, j)\} \rightarrow (k, i, j)$. Notation $P \rightarrow (k, i, j)$ means $p \rightarrow (k, i, j)$ for every $p \in P$, and P is called *a predecessor set* of task (k, i, j). In addition, type II$_j$ task (k, k, j) with $j \neq k$ reads tile A_{kk} written by task (k, k, k). Thus, task (k, k, j) with $j \neq k$ is flow dependent on task (k, k, k). This is summarized as Lemma flow-2.1 in Table II. Similarly, type II$_i$ task (k, i, k) with $i \neq k$ is flow dependent on task (k, k, k) as summarized as Lemma flow-2.2 in Table II. Type III task (k, i, j) with $i \neq k$ and $j \neq k$ reads values $A_{ik}^{(k-1)}$ and $A_{kj}^{(k)}$ from tiles A_{ik} and A_{kj} written by tasks $(k-1, i, k)$ (if $k > 0$) and (k, k, j), respectively. Thus, task (k, i, j) is flow dependent on tasks $(k-1, i, k)$ (if $k > 0$) and (k, k, j). This is summarized as Lemmas flow-3.1 and flow-3.2, respectively, in Table II.

Note that the predecessors shown by Lemma flow-0 and flow-3.2 are from the previous iteration of loop k. The remaining predecessor sets are from the current iteration of loop k.

From the flow dependency predecessors of task (k, i, j) in Table II, the flow dependency successors for task (k, i, j) can be derived. Table III shows the Lemmas for the set of successors of all types of tasks. Lemma flow-D0 is the duel of Lemma flow-0 shown in Table II. Likewise, Lemma flow-D2.1-2 is the duel of Lemma flow-2.1 and flow-2.2. Lemmas flow-D3.1 and flow-D3.2 are the duals of Lemmas flow-3.1 and flow-3.2, respectively.

Table II. Flow dependency predecessors of task (k, i, j).

Lemma	Type	Condition	Set of processors
flow-0	All	$k > 0$	$\{(k-1, i, j)\} \rightarrow (k, i, j)$
flow-2.1	II$_j$	$i = k \wedge j \neq k$	$\{(k, i, k)\} \rightarrow (k, i, j)$
flow-2.2	II$_i$	$i \neq k \wedge j = k$	$\{(k, k, j)\} \rightarrow (k, i, j)$
flow-3.1	III	$i \neq k \wedge j \neq k$	$\{(k, k, j)\} \rightarrow (k, i, j)$
flow-3.2	III	$i \neq k \wedge j \neq k \wedge k > 0$	$\{(k-1, i, k)\} \rightarrow (k, i, j)$

Table III. Flow dependency successors of task (k, i, j).

Lemma	Type	Condition	Set of successors
flow-D0	All	$k < d-1$	$(k,i,j) \rightarrow \{(k+1,i,j)\}$
flow-D2.1-2	I	$i = k \wedge j = k$	$(k,i,j) \rightarrow \{(k,i,y) \mid y \neq k\} \cup \{(k,x,j) \mid x \neq k\}$
flow-D3.1	II_j	$i = k \wedge j \neq k$	$(k,i,j) \rightarrow \{(k,x,j) \mid x \neq k\}$
flow-D3.2	II_i	$i \neq k+1 \wedge j = k+1 \wedge k < d-1$	$(k,i,j) \rightarrow \{(k+1,i,y) \mid y \neq k+1\}$

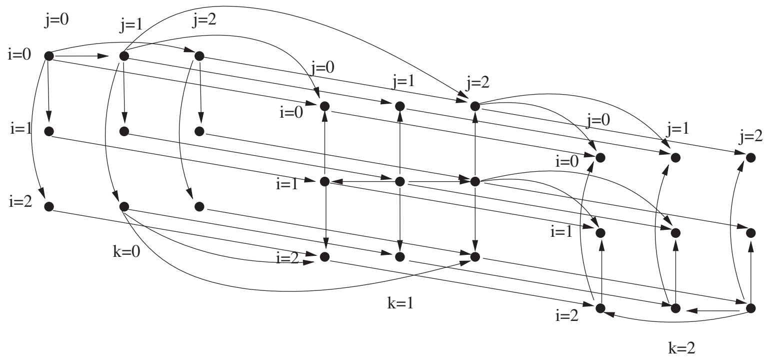
Figure 4(a) shows the flow dependencies in the task graph of the tiled Gauss–Jordan elimination with 3×3 tiles. Each dot is a task of the $3 \times 3 \times 3$ task graph, and arrows are the flow dependencies between the tasks according to Table II or III.

2.2. Anti data dependency

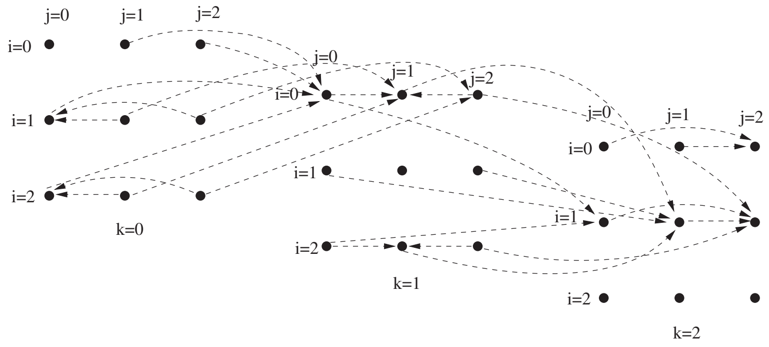
The tasks that a task (k, i, j) is anti dependent on are either of the same iteration k or the previous iteration $k-1$. To find the tasks that task (k, i, j) is anti dependent on, we need to check all the prior tasks that reads A_{ij}, ever since task $(k-1, i, j)$ wrote A_{ij} with value $A_{ij}^{(k-1)}$.

For the type I task (k, k, k), the previous task writing A_{kk} is task $(k-1, k, k)$ with value $A_{kk}^{(k-1)}$. Task $(k-1, k, k)$ is of type III, and no tasks in iteration $k-1$ read the A_{kk} written by task $(k-1, k, k)$. Also, type I task (k, k, k) is the first task in iteration k, and thus, there are no prior tasks in iteration k to read A_{kk}. Therefore, we can conclude that type I task (k, k, k) is not anti dependent on any tasks.

For a type II_i task (k, i, k) with $i \neq k$, the type III tasks in $\{(k, i, y) \mid y \neq k\}$ of the current iteration k read A_{ik} before the task (k, i, k) writes it. Thus, we can say that task (k, i, k) is anti dependent on the tasks in $\{(k, i, y) \mid y \neq k\}$ or $\{(k, i, y) \mid y \neq k\} \mapsto (k, i, k)$. This is summarized



(a) Flow Dependencies of 3 × 3 × 3 Task Graph



(b) Anti Dependencies of 3 × 3 × 3 Task Graph

Figure 4. (a) Flow and (b) anti dependencies of $3 \times 3 \times 3$ task graph.

as Lemma anti-2.1 in Table IV. To find the tasks of the previous iteration $k-1$ that read A_{ik}, we first need to remember that A_{ik} was written previously by task $(k-1, i, k)$ with value $A_{ik}^{(k-1)}$. If $i \neq k-1$, task $(k-1, i, k)$ is of type III. Thus, no tasks in iteration $k-1$ read the values $A_{ik}^{(k-1)}$ from A_{ij}. However, if $i = k-1$, task $(k-1, k-1, k)$ is of type II_j. A_{ik} is read by the type III tasks in $\{(k-1, x, k) \mid x \neq k-1\}$. Thus, $\{(k-1, x, k) \mid x \neq k-1\}$ is a predecessor set of task (k, i, k) with $i = k-1$. This is summarized as Lemma anti-2.2 in Table IV.

For a type II_j task (k, k, j) with $j \neq k$ writing A_{kj}, there are obviously no tasks in the current iteration k reading value $A_{kj}^{(k-1)}$ from A_{kj} except itself. To find possible tasks in iteration $k-1$ that may read value $A_{kj}^{(k-1)}$ from A_{kj}, which is written by task $(k-1, k, j)$, we consider the case of $j \neq k-1$ first. The task $(k-1, k, j)$ is of type III because $j \neq k-1$, and thus, there are no tasks in iteration $k-1$ reading value $A_{kj}^{(k-1)}$ from A_{kj}. If $j = k-1$, then task $(k-1, k, j)$ is of type II_i, and there are no tasks in iteration $k-1$ reading value $A_{kj}^{(k-1)}$ from A_{kj} either. Therefore, type II_j task (k, k, j) with $j \neq k$ is not anti dependent on any tasks.

For a type III task (k, i, j) with $i \neq k \wedge j \neq k$, there are obviously no tasks in the current iteration k reading value $A_{ij}^{(k-1)}$ from A_{ij} except itself. To find possible tasks in iteration $k-1$ that may read value $A_{ij}^{(k-1)}$ from A_{ij}, which is written by task $(k-1, i, j)$, we need to consider the following four cases:

1. $i \neq k-1 \wedge j \neq k-1$. In this case, task $(k-1, i, j)$ writing value $A_{ij}^{(k-1)}$ to A_{ij} is of type III, and no tasks in iteration $k-1$ read $A_{ij}^{(k-1)}$ from A_{ij}.
2. $i \neq k-1 \wedge j = k-1$. In this case, task $(k-1, i, j)$ is of type II_i, and no tasks in iteration $k-1$ read $A_{ij}^{(k-1)}$ from A_{ij} either.
3. $i = k-1 \wedge j \neq k-1$. In this case, task $(k-1, i, j)$ is of type II_j, and the type III tasks in $\{(k-1, x, j) \mid x \neq k-1\}$ read $A_{ij}^{(k-1)}$ from A_{ij}. Therefore, task (k, i, j) with $i = k-1 \wedge j \neq k \wedge j \neq k-1 \wedge k > 0$ is anti dependent on the tasks in $\{(k-1, x, j) \mid x \neq k-1\}$. This is summarized as Lemma anti-3.1 in Table IV.
4. $i = k-1 \wedge j = k-1$. In this case, task $(k-1, i, j)$ is of type I, and the type II_i tasks in $\{(k-1, x, j) \mid x \neq k-1\}$ and the type II_j tasks in $\{(k-1, i, y) \mid y \neq k-1\}$ read value $A_{ij}^{(k-1)}$ from A_{ij} written by task $(k-1, i, j)$. Thus, task (k, i, j) with $i = k-1 \wedge j = k-1 \wedge k > 0$ is anti dependent on the tasks in $\{(k-1, x, j) \mid x \neq k-1\}$ and $\{(k-1, i, y) \mid y \neq k-1\}$. This is summarized as Lemma anti-3.2 in Table IV.

Similar to flow data dependency, anti data dependency successors of each task can be derived from the predecessors in Table IV. The Lemmas for anti data dependency successors of task (k, i, j) are shown in Table V. Lemmas anti-D2.1, anti-D2.2, anti-D3.1, and anti-D3.2 in Table V are the duels of Lemmas anti-2.1, anti-2.2, anti-3.1, and anti-3.2 in Table IV, respectively.

The anti data dependencies of the $3 \times 3 \times 3$ task graph for the same tiled Gauss–Jordan elimination with 3×3 tiles, according to Table IV or V, are shown in Figure 4(b) as dashed arrows.

Table IV. Anti dependency predecessors of task (k, i, j).

Lemma	Type	Condition	Set of processors
anti-2.1	II_i	$i \neq k \wedge j = k$	$\{(k, i, y) \mid y \neq k\} \mapsto (k, i, j)$
anti-2.2	II_i	$i = k-1 \wedge j = k \wedge k > 0$	$\{(k-1, x, j) \mid x \neq k-1\} \mapsto (k, i, j)$
anti-3.1	III	$i = k-1 \wedge j \neq k \wedge j \neq k-1 \wedge k > 0$	$\{(k-1, x, j) \mid x \neq k-1\} \mapsto (k, i, j)$
anti-3.2	III	$i = k-1 \wedge j = k-1 \wedge k > 0$	$\{(k-1, x, j) \mid x \neq k-1\} \cup$ $\{(k-1, i, y) \mid y \neq k-1\} \mapsto (k, i, j)$

Table V. Anti dependency successors of task (k, i, j).

Lemma	Type	Condition	Set of successors
anti-D2.1	III	$i \neq k \wedge j \neq k$	$(k,i,j) \mapsto \{(k,i,k)\}$
anti-D2.2	II_j	$i \neq k \wedge j = k+1 \wedge k < d-1$	$(k,i,j) \mapsto \{(k+1,k,j)\}$
anti-D3.1	III	$i \neq k \wedge j \neq k \wedge j \neq k+1 \wedge k < d-1$	$(k,i,j) \mapsto \{(k+1,k,j)\}$
anti-D3.2	II_i and II_j	$(i \neq k \wedge j = k \vee i = k \wedge j \neq k) \wedge k < d-1$	$(k,i,j) \mapsto \{(k+1,k,k)\}$

2.3. *Output data dependency*

In the tiled Gauss–Jordan elimination algorithm, each task (k, i, j) overwrites A_{ij} written by task $(k-1, i, j)$ (if $k > 0$). Thus, each task (k, i, j) is only output dependent on task $(k-1, i, j)$ (if $k > 0$). Recall that each task (k, i, j) is also flow dependent on task $(k-1, i, j)$ (if $k > 0$) as shown by Lemma flow-0 in Table II. Therefore, the output data dependencies are subsumed by the flow data dependencies, and they need not be enforced.

Finally, the task graph of the tiled Gauss–Jordan elimination for shared-memory models is the graph that consists of all d^3 tasks (k, i, j) $(0 \leqslant k, i, j \leqslant d-1)$, with all the flow data dependency edges determined by Tables II or III and all the anti data dependency edges determined by Tables IV or V. For the 3×3 tiled Gauss–Jordan elimination example, the task graph is obtained by including the edges in both Figures 4(a) and 4(b).

3. IMPLEMENTATION IN CONCURRENT COLLECTIONS

Concurrent Collections is a novel programming model that separates the specification of computation from that of parallelism. Programmers can define computational *steps*, data *items*, and control *tags* in a CnC *context*. Each instance of a step or an item has its unique identifier called *tag*. A *step collection*, the set of instances of a step type, corresponds to exactly one prescribing *tag collection*, and a step instance can execute only if the matching tag of the tag collection becomes available. A *data collection* is the set of instances of a data type that can be assigned only once and cannot be overwritten. A computation step may produce items and tags. It may also need to input data items from the CnC context. When tags and data items are produced by a step, they become available. If a tag prescribing a step becomes available, the step is *prescribed*. If all the data items that a step needs become available, the step is *input-available*. A step is *enabled* if it is both *prescribed* and *input-available*. Only enabled steps may execute. Items and tags may also be input from or output to the *environment*, which is the external code invoking the CnC computation. Details of the CnC programming model can be found in [4, 9, 10].

The implementation of the tiled Gauss–Jordan elimination in CnC described in the succeeding text is from `samples/MatrixInverter/` of the Intel CnC for C++ 0.6 for Linux from [4]. As analyzed in Section 2, there are $d \times d \times d$ tasks (k, i, j) $(0 \leqslant k, i, j \leqslant d-1)$, updating tile A_{ij} to its k-th version $A_{ij}^{(k)}$. In CnC, each data item is a single-assignment object. Therefore, each value $A_{ij}^{(k)}$ is stored in a separate data item. Thus, it is natural that we have the following: (i) a step collection for the tasks (k, i, j) $(0 \leqslant k, i, j \leqslant d-1)$ to calculate output values $A_{ij}^{(k)}$; (ii) a tag collection of triplets (k, i, j) $(0 \leqslant k, i, j \leqslant d-1)$ to prescribe the tasks of the step collection; and (iii) an item collection of all values $A_{ij}^{(k)}$ identified by the triplet tag (k, i, j) $(0 \leqslant k, i, j \leqslant d-1)$. The initial/final tile array is input/output from/to the environment as the tiles of version -1/$d-1$, that is, $A_{ij}^{(-1)}/A_{ij}^{(d-1)}$ $(0 \leqslant i, j \leqslant d-1)$. The environment produces all the tags (k, i, j) $(0 \leqslant k, i, j \leqslant d-1)$ to prescribe all the steps. Because each value $A_{ij}^{(k)}$ is stored in separate data item, all the anti and output data dependencies in the original algorithm in Figure 1 are eliminated. Only the flow dependencies between the steps shown in Tables II or III need to be enforced.

Figure 5 shows the text-format CnC graph for the tiled Gauss–Jordan elimination [4]. It has an item collection named `m_tiles` of data type `tile` (to store values $A_{ij}^{(k)}$) associated with the tag `tile_tag` specified in square brackets, a tag collection named `m_steps` (for indexes

```
// Declarations
[tile m_tiles <tile_tag>];
<tile_tag m_steps>;
(compute_inverse)tuner=my_tuner;

// Relations
env -> [m_tiles], <m_steps>;
<m_steps> :: (compute_inverse);
[m_tiles] -> (compute_inverse) -> [m_tiles];
[m_tiles] -> env;
```

Figure 5. Concurrent collection graph for tiled Gauss–Jordan elimination [4].

(k, i, j)) associated with tags `tile_tag` specified in angle brackets, and a step collection named `compute_inverse` (for tasks) specified in parentheses. The arrow `->` shows the producer and consumer relations between items and steps, and double colon `::` shows the relation of prescription between tags and steps.

The producer and consumer relations in the CnC graph in Figure 5 do not show the detail of flow data dependencies between the tasks. The flow data dependencies between tasks are enforced by calling the `get()` method to input the tile items the step needs before the computation and by calling the `put()` method to output the tiles the step produces after the computation. For example, the step for type III task (k, i, j) with $i \neq k \wedge j \neq k$ would need tile items $A_{ij}^{(k-1)}$, $A_{ik}^{(k-1)}$, and $A_{kj}^{(k)}$ from the item collection `m_tiles` before it can start the computation. It will output tile item $A_{ij}^{(k)}$ to the item collection `m_tiles` after it finishes its computation. Figure 6 shows the algorithm of the step `compute_inverse` in the CnC implementation from [4]. Note that every data item needed/produced by a task has to be copied from/to the CnC context. If any data item needed by a task, which has already started, is not available, the task will be blocked and put back into a local

```
if ($i = k \wedge j = k$) {            //type I task
    tile $t1$ = get($A_{ij}^{(k-1)}$);
    tile $tout = (t1)^{-1}$;
    put($tout$) as $A_{ij}^{(k)}$;
} else if ($i = k$) {                 //type II$_j$ task
    tile $t1$ = get($A_{kk}^{(k)}$);
    tile $t2$ = get($A_{ij}^{(k-1)}$);
    tile $tout = t1 \cdot t2$;
    put($tout$) as $A_{ij}^{(k)}$;
} else if ($j = k$) {                 //type II$_i$ task
    tile $t1$ = get($A_{ij}^{(k-1)}$);
    tile $t2$ = get($A_{kk}^{(k)}$);
    tile $tout = -t1 \cdot t2$;
    put($tout$) as $A_{ij}^{(k)}$;
} else {                              //type III task
    tile $t1$ = get($A_{ij}^{(k-1)}$);
    tile $t2$ = get($A_{ik}^{(k-1)}$);
    tile $t2$ = get($A_{kj}^{(k)}$);
    tile $tout = t1 - t2 \cdot t3$;
    put($tout$) as $A_{ij}^{(k)}$;
}
```

Figure 6. Algorithm of task (k, i, j) in Concurrent Collections [4].

queue. The blocked task will be rescheduled to the ready queue of the thread when it gets all the data items it needs [17]. This is obviously inefficient if the task has to be rescheduled many times. So, CnC provides a `tuner` [4] class, which allows programmers to spell out the data items that a task needs so that the scheduler will not schedule the task until all the data items it needs are available. We run CnC code both with and without the tuner in our experimentation to see the difference between their performances.

Another important optimization in CnC is the memory management of single-assignment data items [18]. Because CnC does not know how many steps (tasks) will read a data item (through the `get()` method), it will keep the data item indefinitely, and the memory usage would be exploded. CnC allows programmers to pass the number of the tasks that would read the data item, called the reference count, as an argument to the `put()` method when the data item is output to the context. The reference count will be decremented whenever `get()` is called. The storage of the data item is garbage collected when its reference count reaches zero. We always turn this optimization on when we run the CnC code. The details of the CnC implementation of tiled Gauss–Jordan elimination can be found in [4].

4. IMPLEMENTATION IN THREADING BUILDING BLOCKS

Threading Building Blocks allows one to build and evaluate task graphs by using the reference count, `ref_count`, of the task class of TBB. In particular, the task graph can be built by

1. Setting up the `ref_count` of each task to the number of its predecessors.
2. Inserting the code to decrement the `ref_count` of each of its successors towards the end of the task and spawn the successor if its `ref_count` reaches zero after the decrement.

Spawning a task is to put it in the task queue of the physical thread. The TBB scheduler will pick up the tasks in the ready queue of the physical thread to run. TBB uses work-stealing scheduling, and load balance is achieved by letting idle threads steal tasks from the ready queues of other physical threads.

We can add an additional empty task to the graph to be the successor of all the tasks that do not have successors in the original graph. To evaluate and execute the graph, the main driver simply spawns the tasks that do not have predecessors and wait on the added empty task. The tasks will be executed in the order prescribed by the edges of the task graph because a task will not be put in the task queue of a physical thread until its `ref_count` becomes zero, and its `ref_count` reaches zero only if all of its predecessor tasks are completed.

Because TBB accesses and updates the tile array A_{ij} $(0 \leqslant i, j \leqslant d-1)$ directly, the task graph has to include all the flow, anti, and output data dependencies described in Section 2. For this Gauss–Jordan elimination algorithm, the output data dependencies are subsumed by the flow data dependencies, and we only need to include the flow and anti data dependencies in the task graph.

The implementation has two classes: `task_graph` for the task graph and `DagTask` for the task as a subclass of TBB task class `tbb::task`. The `build-graph()` method (of the `task_graph` class) is to build the task graph. Its pseudocode is shown in Figure 7. It basically creates d^3 tasks (k, i, j) $(0 \leqslant k, i, j \leqslant d-1)$ referenced by $m\_tasks[k][i][j]$. For each task, it calculates the number of flow and anti data dependency predecessors according to the Lemmas in Tables II and IV and stores them in variables `flow_dep_count` and `anti_dep_count`, respectively. It finally sets the `ref_count` of the task $m\_tasks[k][i][j]$ with the sum of `flow_dep_count` and `anti_dep_count`.

The computation of a TBB task is specified in its `execute()` method. Pseudocode of the `execute()` method of the `DagTask` task is shown in Figure 8. It first executes the computation according to the type of the task determined by its index (k, i, j) as summarized in Table I. The task reads the tiles that it needs and writes the tile it calculates directly. There is no copying of tiles back and forth as in the CnC implementation. The second part of `execute()` is to decrement the `ref_count` of each of its flow and anti data dependency successors determined by the Lemmas in

```
for $k = 0$ to $d-1$
  for $i = 0$ to $d-1$
    for $j = 0$ to $d-1$ {
      $m\_tasks[k][i][j]$ = new DagTask;
      int flow_dep_count = 0;
      if ($k > 1$) flow_dep_count++;                    //Lemma flow-0
      if ($i = k \wedge j \neq k \vee i \neq k \wedge j = k$)
        flow_dep_count++;                               //Lemmas flow-2.1-2
      if ($i \neq k \wedge j \neq k$) {
        flow_dep_count++;                               //Lemma flow-3.1
        if ($k > 0$) flow_dep_count++;                  //Lemma flow-3.2
      }
      int anti_dep_count = 0;
      if ($i \neq k \wedge j = k$)
        anti_dep_count+= $d-1$;                         //Lemma anti-2.1
      if ($i = k-1 \wedge k > 0$) {
        if ($j = k$) anti_dep_count += $d-1$;           //Lemma anti-2.2
        if ($j = k-1$) anti_dep_count += $2(d-1)$;      //Lemma anti-3.2
        if ($j \neq k \wedge j \neq k-1$)
          anti_dep_count += $d-1$;                      //Lemma anti-3.1
      }
      $m\_tasks[k][i][j]$->ref_count =
        flow_dep_count + anti_dep_count;
    }
```

Figure 7. Algorithm of `build-graph()` of `task_graph`.

Tables III and V, respectively. If the `ref_count` of any successor is zero after the decrement, the task spawns that successor, putting it in the task queue of the physical thread.

5. OVERHEAD OF CONCURRENT COLLECTIONS OVER THREADING BUILDING BLOCKS FOR GAUSS–JORDAN ALGORITHM

To measure the overhead of CnC over TBB, we run the TBB and CnC implementations on an Octal-core AMD Opteron(tm) 2.8 GHz 8220 processor with 64 GByte memory and 1024KB cache and 1024 TLB for 4K pages in each core. We vary the number of cores to use from 1 to 8 for problem sizes from 1024 to 5120. We also run the sequential tiled Gauss–Jordan elimination in Figure 1 as the base sequential time in calculating speedups of parallel executions. To see the impact of the tile size on performance and find the best tile size, we vary the tile size from 8 to 256. For each configuration, we run the code five times and take the average of the execution times as its execution time.

The execution times of problem size 2048 with different tile sizes t ranging from $8, 16, \cdots$ to 256 of TBB and CnC with and without tuner are shown in Figures 9(a), 9(b), and 9(c), respectively.

We calculated the GFLOPS by dividing the number of floating-point operations flop(n,t) by the execution time as follows:

$$\text{GFLOPS}(n,t) = \frac{\text{flop}(n,t)}{10^9 \times e(n,t)} \tag{3}$$

where flop(n,t) is as shown in (1) and $e(n,t)$ is the execution time in seconds for problem size n and tile size t. The GFLOPS for TBB, CnC with tuner, and CnC without tuner for problem size 2048 is shown in Figures 10(a), 10(b), and 10(c), respectively.

Figure 11 shows the speedup over the sequential execution time calculated by the following:

$$S_{\text{np}}(n,t) = \frac{e_{\text{s}}(n,t)}{e_{\text{np}}(n,t)} \tag{4}$$

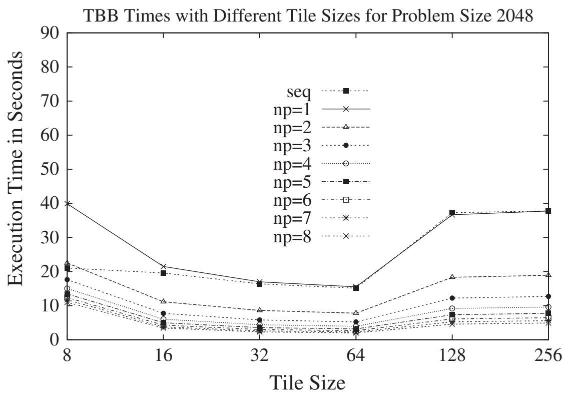
where $e_{\text{s}}(n,t)$ is the execution time of the sequential algorithm in Figure 1 and $e_{\text{np}}(n,t)$ is the parallel execution for problem size n and tile size t. The speedups of TBB, CnC with tuner, and CnC without tuner for problem size 2048 are shown in Figures 11(a), 11(b), and 11(c), respectively.

```
$dec\_spawn\_at\_zero(k, i, j)$ {
  if ($m\_tasks[k][i][j]$->$decrement\_ref\_count()$==0)
    $spawn(m\_tasks[k][i][j])$;
}

execute() {
  if ($i = k \wedge j = k$) {                              //type I task
    $A_{ij} = (A_{ij})^{-1}$;
  } else if ($i = k \wedge j \neq k$) {                    //type II$_j$ task
    $A_{kj} = A_{kk}A_{kj}$;
  } else if ($i \neq k \wedge j = k$) {                    //type II$_i$ task
    $A_{ik} = -A_{ik}A_{kk}$;
  } else {                                                 //type III task
    $A_{ij} = A_{ij} - A_{ik}A_{kj}$;
  }
  // decrement flow dependency successors
  if ($k < d - 1$) $dec\_spawn\_at\_zero(k + 1, i, j)$;    //Lemma flow-D0
  if ($i = k \wedge j = k$) {
    for $y = 0$ to $d - 1$ {
      if ($y = k$) continue;
      $dec\_spawn\_at\_zero(k, i, y)$
    }
    for $x = 0$ to $d - 1$ {
      if ($x = k$) continue;
      $dec\_spawn\_at\_zero(k, x, j)$
    }
  }                                                        //Lemma flow-D2.1-2
  if ($i = k \wedge j \neq k$)
    for $x = 0$ to $d - 1$ {
      if ($x = k$) continue;
      $dec\_spawn\_at\_zero(k, x, j)$
    }                                                      //Lemma flow-D3.1
  if ($i \neq k + 1 \wedge j = k + 1 \wedge k < d - 1$)
    for $y = 0$ to $d - 1$ {
      if ($y = k + 1$) continue;
      $dec\_spawn\_at\_zero(k + 1, i, y)$
    }                                                      //Lemma flow-D3.3

  // decrement anti dependency successors
  if ($i \neq k \wedge j \neq k$)
    $dec\_spawn\_at\_zero(k, i, k)$;                        //Lemma anti-D2.1
  if ($k < d - 1$) {
    if ($i \neq k \wedge j = k + 1$)
      $dec\_spawn\_at\_zero(k + 1, k, j)$;                  //Lemma anti-D2.2
    if ($i \neq k \wedge j \neq k \wedge j \neq k + 1$)
      $dec\_spawn\_at\_zero(k + 1, k, j)$;                  //Lemma anti-D3.1
    if ($i \neq k \wedge j = k \vee i = k \wedge j \neq k$)
      $dec\_spawn\_at\_zero(k + 1, k, k)$;                  //Lemma anti-D3.2
  }
}
```

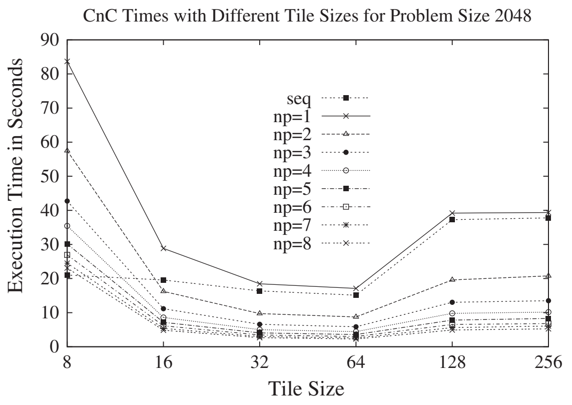
Figure 8. Algorithm of `execute()` for task (k, i, j) in Threading Building Blocks.

5.1. Performance analysis

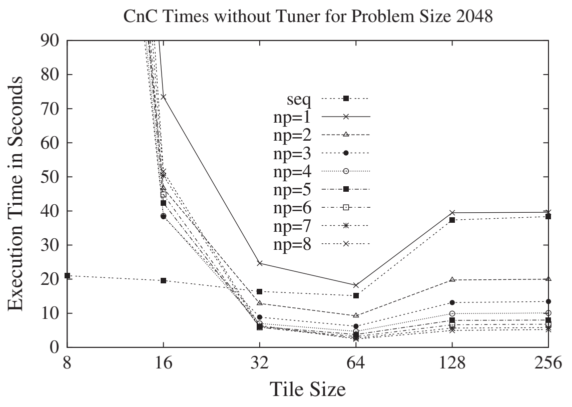
We first observed in Figures 9(a), 9(b), and 9(c) that the sequential execution time `seq` (i.e., $e_s(n, t)$) decreases as tile size t increases from 8 to 64. It then increases sharply from 15.12 to 37.26 (2.46 fold increase) when t changes from 64 to 128, although the number of floating-point operations decreases as the tile size increases according to (1). This is because of the degradation of cache hit ratio when the tile size crosses from 64 to 128 [11]. This is true for all other problem sizes. We



(a) TBB Execution Time



(b) CnC with Tuner Execution Time

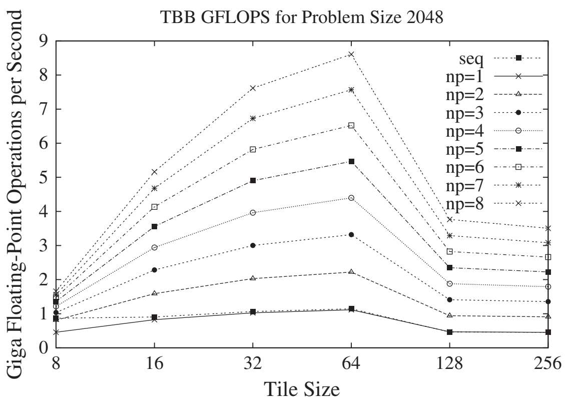


(c) CnC without Tuner Execution Time

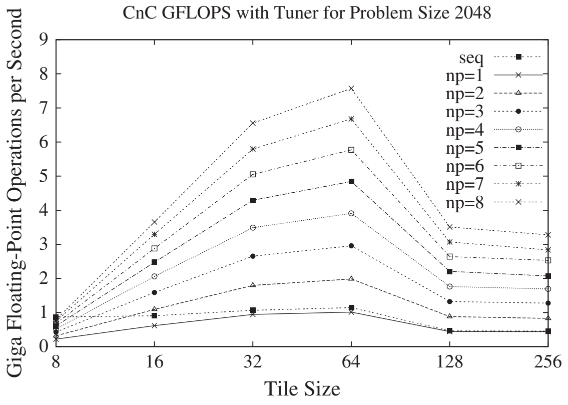
Figure 9. Execution time for problem size 2048. (a) Threading Building Blocks (TBB) execution time, (b) Concurrent Collections (CnC) with tuner execution time, and (c) CnC without tuner execution time.

calculated the ratio $\frac{e(n,128)}{e(n,64)}$ for all problem sizes $n = 1024,\ 2048,\ 3074,\ 4096,$ and 5120, and they are all around 2.46. This confirms the cache miss analysis in [11] and gives us 64 as the best tile size for our machine.

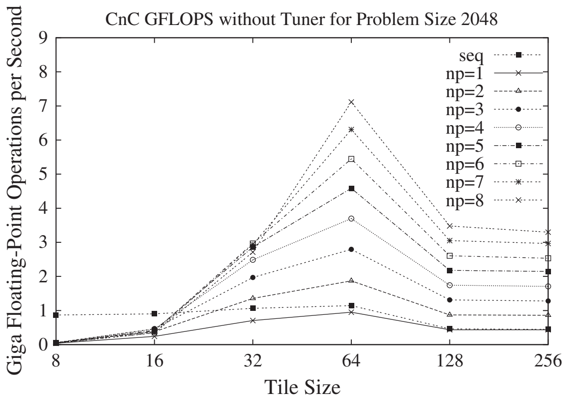
The `np=1` time in Figures 9(a), 9(b), and 9(c) is the parallel execution time using one core. The gap between the `np=1` time and the sequential time `seq` is the overhead of the parallel code over the sequential code. We observed that the smaller the tile size, the larger this gap. This is because smaller tile sizes generate more tasks in the task graph that need to be scheduled. The gap between



(a) TBB GFLOPS



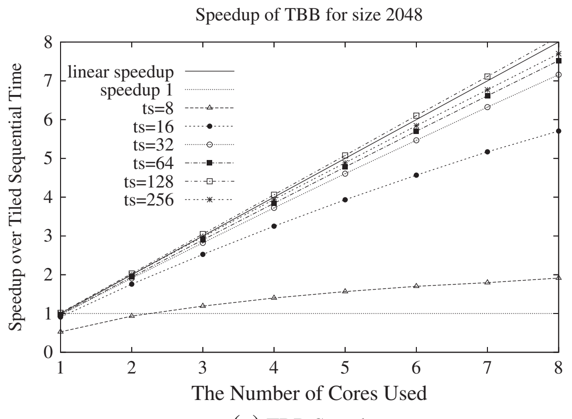
(b) CnC With Tuner GFLOPS



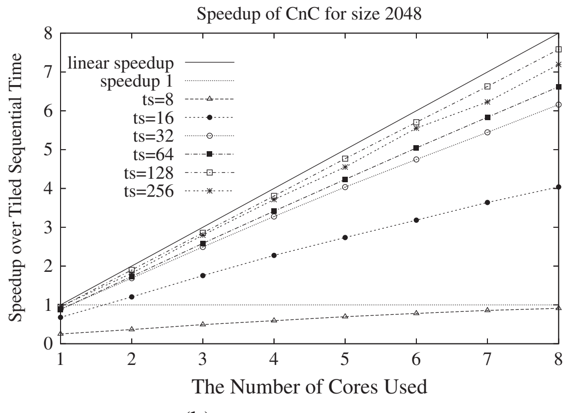
(c) CnC Without Tuner GFLOPS

Figure 10. Giga floating-point operations per second (GFLOPS) performance for problem size 2048. (a) Threading Building Blocks (TBB) GFLOPS, (b) Concurrent Collections (CnC) with tuner GFLOPS, and (c) CnC without tuner GFLOPS.

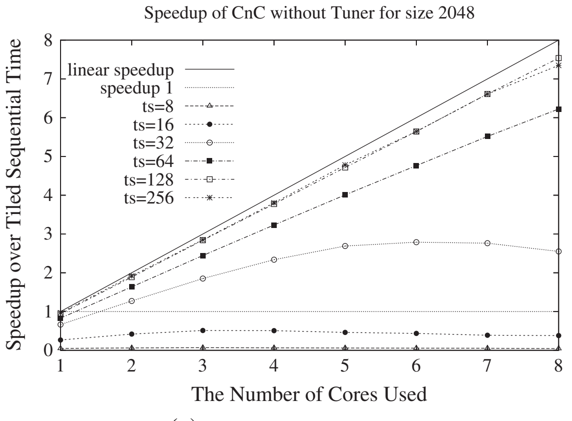
`np=1` time and `seq` time in CnC is larger than TBB (see Figs. 9(a) and 9(b)). For the CnC without tuner, this gap is so large that all the parallel execution times using $1, 2, \cdots, 8$ cores for tile sizes 8 and 16 are greater than the sequential time (see Figure 9(c)). Also, notice that for tile size 128, TBB time using one core is a bit less than the sequential time (see Figure 9(a)). This may be because of the fact that TBB uses its own efficient memory allocator [5, 6, 19], and this is the reason for the super-liner speedup of TBB for tile size 128 (see Figure 11(a)).



(a) TBB Speedup



(b) CnC with Tuner Speedup



(c) CnC without Tuner Speedup

Figure 11. Speedup for problem size 2048. (a) Threading Building Blocks (TBB) speedup, (b) Concurrent Collections (CnC) with tuner speedup, and (c) CnC without tuner speedup.

The speedups with tiles sizes 128 and 256 are higher than the tile size 64. The reason is that poor cache performance makes the task execution time 2.46 times greater and that amortizes the scheduling overhead of the parallel execution. But, the tile sizes 128 and 256 have very low GFLOPS (see Figures 10(a), 10(b), and 10(c)).

For the optimal tile size 64, the speedup over sequential time for TBB, CnC with tuner, and CnC without tuner using eight cores are 7.52, 6.62, and 6.22, respectively. The GFLOPS performances of

TBB, CnC with tuner, and CnC without tuner using eight cores are 8.61, 7.57, and 7.12 GFLOPS, respectively.

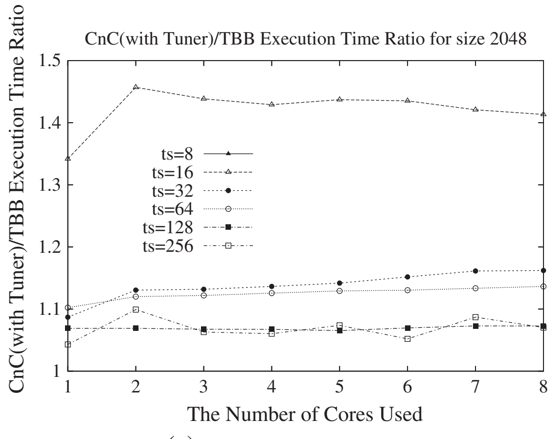
5.2. *Measuring the overhead of Concurrent Collections*

To measure the overhead of CnC over TBB, we calculated the CnC/TBB ratio as follows:

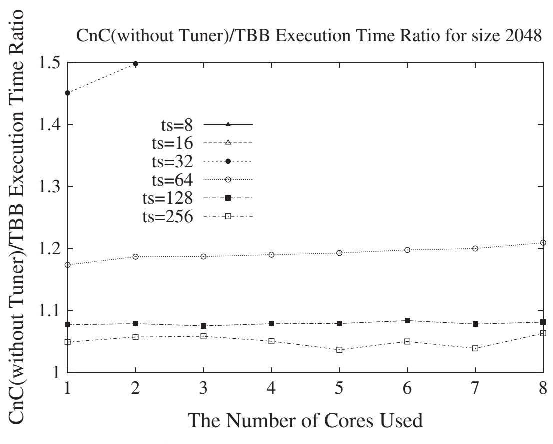
$$R_{\mathrm{CnC/TBB}} = \frac{e_{\mathrm{CnC}}(n,t)}{e_{\mathrm{TBB}}(n,t)} \tag{5}$$

where $e_{\mathrm{CnC}}(n,t)$ and $e_{\mathrm{TBB}}(n,t)$ are the CnC and TBB execution times, respectively.

The CnC/TBB ratios, $R_{\mathrm{CnC/TBB}}$, for different tile sizes and problem size $n = 2048$ are plotted in Figure 12. Figures 12(a) and 12(b) are the ratios of the CnC with and without tuner over TBB, respectively. The CnC/TBB ratios for tile size 8, 16, and 32 without tuner are very large and out of the y-range in the plot in Figure 12(b). For the optimal tile size 64, the CnC/TBB ratio is increasing with the number of cores used. For the CnC with tuner, the CnC/TBB is between 1.10 (using 1 core) and 1.14 (using 8 cores), or the overhead CnC over TBB is between 10% and 14% of TBB time (see Figure 12(a)). For the CnC without tuner, the CnC/TBB ratio for tile size 64 is between 1.17 (using 1 core) and 1.21 (using 8 cores), with the CnC overhead between 17% and 21% of TBB time (see Figure 12(b)).

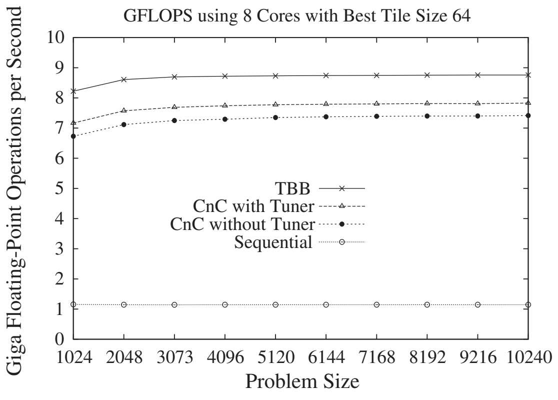


(a) CnC (withTuner)/TBB Ratio

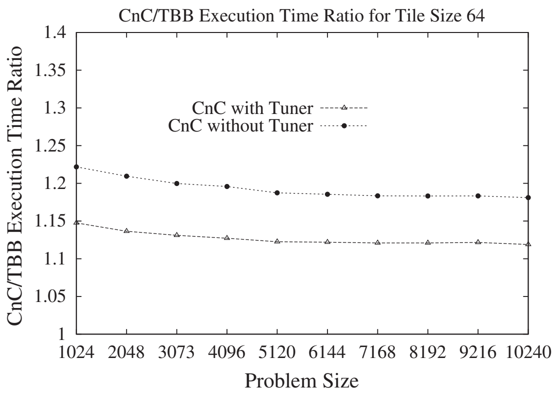


(b) CnC (withoutTuner)/TBB Ratio

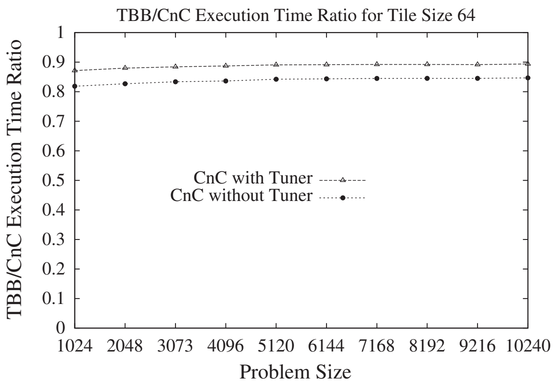
Figure 12. Concurrent Collections (CnC)/Threading Building Blocks (TBB) execution time ratio for problem size 2048. (a) CnC (with tuner)/TBB ratio and (b) CnC (without tuner)/TBB ratio.



(a) GFLOPS using 8 Cores with Tile Size 64



(b) CnC/TBB Ratio using 8 Cores with Tile Size 64



(c) TBB/CnC Ratio using 8 Cores with Tile Size 64

Figure 13. Performance of the optimal tile size 64 using eight cores. (a) Giga floating-point operations per second (GFLOPS) using eight cores with tile size 64, (b) Concurrent Collections (CnC)/Threading Building Blocks (TBB) ratio using eight cores with tile size 64, and (c) TBB/CnC ratio using eight cores with tile size 64.

Because the tile size 64 gives the best performance, we run the TBB, CnC with and without tuner implementations, as well as the sequential code using eight cores for the problem sizes of 6144–10240 with fixed tile size 64.

We calculated their GFLOPS according to (3) and plotted the GFLOPS for tile size 64 for all the problem sizes of 1024–10240 in Figure 13(a). We also calculated the CnC/TBB ratios according to (5), which is equivalent to $\frac{\text{GFLOPS}_{\text{TBB}}(n,t)}{\text{GFLOPS}_{\text{CnC}}(n,t)}$ and plotted them in Figure 13(b). Figure 13(a) shows that the GFLOPS of TBB is consistently higher than those of CnC. All the GFLOPS increase with the increasing problem size. Figure 13(b) shows that the CnC/TBB time ratio is between 1.15 and 1.12 for the CnC with tuner and between 1.22 and 1.18 for the CnC without tuner. All the CnC/TBB ratios decrease as the problem size increases. The overhead of CnC over TBB is between 12% and 15% with tuner and between 18% and 22% without tuner.

We also calculated $\frac{\mathrm{GFLOPS}_{\mathrm{CnC}}(n,t)}{\mathrm{GFLOPS}_{\mathrm{TBB}}(n,t)}$, which is the reciprocal of the CnC/TBB ratio, called the TBB/CnC ratio and plotted them in Figure 13(c). The TBB/CnC ratio or $\frac{\mathrm{GFLOPS}_{\mathrm{CnC}}(n,t)}{\mathrm{GFLOPS}_{\mathrm{TBB}}(n,t)}$ shows the percentage of the TBB performance that CnC can deliver. Figure 13(c) shows the CnC with tuner that can deliver as much as 87%-89% of the TBB performance and the CnC without tuner that delivers 82%-85% of the TBB performance.

6. RELATED AND FUTURE WORK

Comprehensive performance evaluation of CnC is reported in [16]. It compares the performance of CnC with that of OpenMP with Math Kernel Library (MKL), Cilk, Multi-threaded MKL, ScaLAPACK with shared memory Message Passing Interface (MPI), and PLASMA with MKL on Cholesky factorization and Eigensolver problem. It shows that CnC is comparable with PLASMA with MKL, which also uses the task graph-based approach of parallel execution, and CnC is superior to all the loop-based OpenMP with MKL, Cilk, Multi-threaded MKL and ScaLAPACK with shared memory MPI. In this paper, we compared both task-based CnC and TBB and showed that TBB is 12%–15% faster than CnC on Gauss–Jordan elimination.

The parallel Gauss–Jordan elimination reported in [12] is loop-based like the one in Figure 3. The work of [16] already showed that loop-based parallel algorithms are always inferior to the task-based algorithm in CnC on the Cholesky factorization and the Eigensolver problem. In this paper, we showed that the task-based algorithm in TBB is about 12%–15% faster than CnC on the Gauss–Jordan elimination. The superior performance of task-based algorithms to the loop-based algorithms in LAPACK or MKL is also reported in [14] on QR factorization and in [15] on Cholesky factorization.

Both parallel tiled Gauss–Jordan algorithms reported in [12] and [13] are restricted on the augmented system using two data arrays. Our parallel Gauss–Jordan algorithm is an in-place algorithm storing the inverted matrix in the same tiled data array of the input matrix.

Our experiments also confirm the analysis of the impact of tile size on cache performance in [11] and found the best tile size for our machine.

Parallel programming of the loop-based parallel Gauss–Jordan algorithm shown in Figure 2 in TBB will be much simpler than the task-based TBB programming shown in this paper. The task-based programming in CnC is also much simpler than the task-based programming in TBB. One future work would be to implement and compare the performances of the task-based parallel Gauss–Jordan algorithm in CnC with a loop-based one in TBB.

Another future work is to compare the performances of Gauss–Jordan elimination in TBB, CnC, and OpenMP (both task-based and loop-based) because OpenMP now supports task-based programming as well.

We also plan to test the scalability of CnC and TBB for parallel Gauss–Jordan elimination using Intel processors with larger number of cores in the future.

7. CONCLUSION AND DISCUSSION

We make the following contributions in this paper: (i) we provided a complete data dependency analysis for the tiled in-place Gauss–Jordan elimination algorithm to guide task-based parallelization; (ii) we parallelized and implemented the tiled in-place Gauss–Jordan elimination in TBB; and (iii) we compared the task-based parallel performances of CnC and TBB and found that the overhead of CnC, which is implemented on top of TBB, is only within 12%–15% of the TBB execution time for the CnC with tuner and 18%–22% of the TBB execution time for the CnC without tuner.

The overheads of CnC over TBB mentioned previously can be translated to that the CnC with tuner delivers 87%–89% of the TBB performance, and the CnC without tuner delivers 82%–85% of the TBB performance.

Task-based parallelizing and programming in CnC is easier for two reasons. Firstly, CnC programming does not need to analyze anti and output dependencies because its data flow single-assignment model eliminates all of them. Secondly, The flow dependencies are enforced and specified through the data items that carry the data flow. This can be simply done by calling the `get()` method for each data item that the task needs. Using tuner in CnC is not difficult either: all that is needed is to specify all the data items the task needs. The data items that a task needs can be easily identified from the code of the task. (However, the memory management optimization of CnC [18] requires one to know the number of tasks that would read the data item, which is the same as the number of flow data dependency successors of the task writing the item. This requires a flow dependency analysis.) In contrast, task-based parallelizing and programming in TBB requires a full data dependency analysis, including the output and especially the complicated anti data dependency analysis.

Given that CnC with tuner can still deliver 87%–89% of the TBB performance and TBB requires the complex data dependency analysis, we believe that CnC is very promising in winning acceptance by parallel programmers for dense linear algebra algorithms in the future.

ACKNOWLEDGEMENT

The work is supported in part by equipment purchased under NASA Award NCC5-597.

REFERENCES

1. Anderson E, Bai Z, Bischof C, Blackford S, Demmel J, Dongarra J, Du Croz J, Greenbaum A, Hammarling S, McKenney A, Sorensen D. *LAPACK User's Guide*. SIAM: Philadephia, PA, 1999.
2. Agullo E, Hadri B, Ltaief H, Dongarrra J. Comparative study of one-sided factorizations with multiple software packages on multi-core hardware. In P*roceedings of the Conference on High Performance Computing Networking, Storage and Analysis* SC '09. ACM: New York, NY, USA, 2009; 20:1–20:12.
3. Haidar A, Ltaief H, YarKhan A, Dongarra J. Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures. *Technical Report Tech. Rep. ut-cs-11-666*, Innovative Computing Laboratory, University of Tennessee, 2011.
4. CnC Team. Intel Concurrent Collections for C++. *Technical Report*, Intel Inc., 2009. http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc/.
5. Reinders J. Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism, Oreilly Associates, 2007.
6. Reinders J. TBB 3.0: New (today) version of Intel Threading Building Blocks. *Technical Report*, Intel Inc., May 2010. http://software.intel.com/en-us/blogs/2010/05/04/tbb-30-new-today-version-of-intel-threading-building-blocks/.
7. Arvind, Gostelow KP, Plouffe W. An asynchronous programming language and computing machine. *Technical Report Tech. Rep. TR 114a*, University of California at Irvine, 1978.
8. Carriero N, Gelernter D. Linda in context. *Communications of the ACM* April 1989; **32**:444–458.
9. Knobe K. Ease of use with Concurrent Collections (CnC). In *Proceedings of the First USENIX conference on Hot topics in parallelism,* HotPar'09. USENIX Association: Berkeley, CA, USA, 2009; Article 17.
10. Chandramowlishwaran A, Knobe K, Vuduc R. Applying the concurrent collections programming model to asynchronous parallel dense linear algebra. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming,* PPoPP '10. ACM: New York, NY, USA, 2010; 345–346.
11. Park N, Hong B, Prasanna VK. Tiling, block data layout, and memory hierarchy performance. *IEEE Transactions on Parallel and Distributed Systems* July 2003; **14**:640–654.
12. Melab N, Talbi E-G, Petiton S. A parallel adaptive Gauss-Jordan algorithm. *Journal of Supercomputing* September 2000; **17**:167–185.
13. Shang L, Petiton S, Hugues M. A new parallel paradigm for block-based Gauss-Jordan algorithm. In *Proceedings of the 2009 International Conference on Grid and Cooperative Computing*, 2009; 193–200.
14. Buttari A, Langou J, Kurzak J, Dongarra J. Parallel tiled QR factorization for multicore architectures. *Concurrency and Computation: Practice and Experience* September 2008; **20**:1573–1590.
15. Buttari A, Langou J, Kurzak J, Dongarra J. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing* January 2009; **35**:38–53.
16. Chandramowlishwaran A, Knobe K, Vuduc R. Performance evaluation of concurrent collections on high-performance multicore computing systems. In *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, April 2010; 1–12.

17. Chandramowlishwaran A, Knobe K, Lowney G, Sarkar V, Treggiari L. Multi-core implementations of the concurrent collections programming model. In *The 14th Workshop on Compilers for Parallel Computing (CPC)*, January 2009.
18. Budimlic Z, Chandramowlishwaran AM, Knobe K, Lowney GN, Sarkar V, Treggiari L. Declarative aspects of memory management in the concurrent collections parallel programming model. In *Proceedings of the 4th workshop on Declarative aspects of multicore programming,* DAMP '09. ACM: New York, NY, USA, 2008; 47–58.
19. Kukanov A, Voss M. The foundations for scalable multi-core software in Intel Threading Building Blocks. *Intel Technology Journal* November 2007; **11**:309–322.