Measuring the Overhead of Intel C++ CnC over TBB for Gauss-Jordan Elimination

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

Abstract

The most efficient parallel execution of dense linear algebra algorithms is to build and evaluate the task graph constrained only by the data dependencies between the tasks. Both Intel C++ Concurrent Collections (CnC) and Threading Building Blocks (TBB) libraries allow such task-based parallel programming. In this paper, we first analyze all the three types of data dependencies in the tiled in-place Gauss-Jordan elimination algorithm and implemented it in TBB. We compare the performances of TBB and CnC, which is built on top of TBB, and found that the overhead of CnC over TBB is only 12% to 15% of the TBB time, and CnC can deliver 87% to 89% of the TBB performance for Gauss-Jordan elimination, using the optimal tile size.

1 Introduction

The most efficient way of parallel execution of dense linear algebra algorithms is to build and evaluate the task graph constrained only by the data dependencies between the tasks. Both Intel C++ Concurrent Collections (CnC) [1] and Threading Building Blocks (TBB) [2] libraries allow such task-based parallel programming. CnC also adapts the macro data-flow model by allowing only single-assignment data objects in its Linda-tuple-space-like [3] global space called CnC context. Each computational task in CnC is purely functional and side-effect free. Therefore, only data-flow dependencies between the tasks need to be honored and they can be enforced through accessing the single-assignment data objects in the CnC context. This makes parallel programming in CnC much easier [4, 5].

Intel C++ CnC library for shared-memory multi-core processors is implemented on top of Intel Threading Building Blocks (TBB) library [2]. TBB is not functional and tasks in TBB read and write the shared memory of multi-core computers directly. Implementing the data-flow model on shared memory may incur overheads 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 objects 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 to determine whether or not to adapt the new programming model or language. We are interested in finding 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.

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 compared the TBB performance with the performance of the CnC implementation [1]. We found that the overhead of CnC over TBB on Gauss-Jordan elimination is only 12% to 15% of the TBB time for the CnC with tuner and 18% to 22% of the TBB time for the CnC without tuner, using the optimal tile size. Given that performance is proportional to the reciprocal of execution time, CnC with tuner and without tuner can thus deliver as much as 87% to 89% and 82% to 85% of the TBB 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 Gauss-Jordan algorithm. Section 3 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 4 presents our experimental results of both CnC and TBB parallel codes, as well as the sequen-
Gauss-Jordan elimination is the algorithm to invert a matrix \( A \) to \( A^{-1} \). The sequential tiled Gauss-Jordan elimination parallelized in \([6, 7]\) inverts an originally identity matrix \( B \) and has \( A^{-1} \) stored in \( B \) when the algorithm is finished. A modified algorithm that inverts \( A \) in place and stores \( A^{-1} \) in \( A \) is shown in Figure 1. Here, an \( n \times n \) data matrix array is tiled to a \( d \times d \) tile array. Each tile is a \( t \times t \) data array with \( t = n/d \) and denoted as \( A_{ij} \) (\( 0 \leq i, j \leq d - 1 \)). 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 \( A_{ij}^{(-1)} \) and the final inverted matrix is the collection of \( A_{ij}^{(d-1)} \).

For \( k = 0, d-1 \)
\[
A_{kk}^{(k)} = (A_{kk}^{(k-1)})^{-1}
\]
(task type I)

For \( j = 0, d-1 \) \& \( j \neq k \)
\[
A_{ij}^{(k)} = A_{kk}^{(k)} A_{kj}^{(k-1)}
\]
(task type II)

For \( i = 0, d-1 \) \& \( i \neq k \)
\[
A_{ij}^{(k)} = A_{ij}^{(k-1)} - A_{ik}^{(k-1)} A_{kj}^{(k)}
\]
(task type III)

Table 1 shows the classification of task types according to their indexes \((k, i, j)\).

<table>
<thead>
<tr>
<th>Type</th>
<th>Condition</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>( i = k ) &amp; ( j = k )</td>
<td>( A_{kk}^{(k)} = (A_{kk}^{(k-1)})^{-1} )</td>
</tr>
<tr>
<td>II (_i)</td>
<td>( i = k ) &amp; ( j \neq k )</td>
<td>( A_{kj}^{(k)} = A_{kk}^{(k)} A_{kj}^{(k-1)} )</td>
</tr>
<tr>
<td>II (_i)</td>
<td>( i \neq k ) &amp; ( j = k )</td>
<td>( A_{ik}^{(k)} = -A_{ik}^{(k-1)} A_{kj}^{(k)} )</td>
</tr>
<tr>
<td>III</td>
<td>( i \neq k ) &amp; ( j \neq k )</td>
<td>( A_{ij}^{(k)} = A_{ij}^{(k-1)} - A_{ik}^{(k-1)} A_{kj}^{(k)} )</td>
</tr>
</tbody>
</table>

Table 1: Types and Index Conditions of Tasks \((k, i, j)\)

There are three kinds of data dependencies between tasks: 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 \( A \rightarrow B(A \rightarrow B) \), 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. Anti and output data dependencies can be eliminated by re-naming data objects as in the single-assignment programming models such as CnC or functional programming languages. For the shared memory models such as TBB, the task graph has to include all the flow, anti and output data dependencies. 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 \( A_{ij} \) \((0 \leq i, j \leq d-1)\) 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 2, 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 \((k, i, j)\). The predecessor sets of tasks of types \( \Pi \_i \) and \( \Pi \_i \) are shown in Lemmas flow-2.1 and flow-2.2 in Table 2, respectively. The predecessor sets of tasks of type \( \Pi \_l \) are shown in Lemmas flow-3.1 and flow-3.2. Note that the predecessors shown by Lemma flow-0 and flow-3.2 are from the previous iteration of loop \( k \).

1The floating-point operations includes add, subtract, multiply and divide between two floating-point numbers.
The predecessor sets shown by Lemmas flow-2.1 and flow-2.2 are from the current iteration of loop \( k \).

From the flow dependency predecessors of task \((k, i, j)\) in Table 2, the flow dependency successors for task \((k, i, j)\) can be derived. Table 3 shows the Lemmas about the set of successors of each task \((k, i, j)\). Lemma flow-D0 is the duel of Lemma flow-2.1 shown in Table 2. Likewise, Lemma flow-D2.1-2 is the duel of Lemma flow-2.2. Lemmas flow-D3.1 and flow-D3.2 are the duals of Lemmas flow-3.1 and flow-3.2, respectively.

### 2.2 Anti Data Dependency

The tasks that a task \((k, i, j)\) is anti-dependent on either 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 \(A_{ij}\) was written by task \((k - 1, i, j)\) 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 \(k\). Thus, there are no tasks in iteration \(k\) reading \(A_{kk}\) except task \((k, k, k)\) itself. Therefore, we can conclude that type I task \((k, k, k)\) is not anti-dependent on any tasks.

For a type II 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 \(\{(k, i, y) \mid y \neq k\} \rightarrow (k, i, k)\). This is summarized as Lemma anti-2.1 in Table 4. To find the tasks of the previous iteration \(k - 1\) that read \(A_{ik}\), we first need to remember that \(A_{ik}\) is 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 \((k - 1, k, 1)\) is of type II. \(A_{ik}\) 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 4.

For a type II task \((k, j, k)\) with \(j \neq k\) writing \(A_{kj}\), there is obviously no tasks in the current iteration \(k\) reading value \(A_{kj}^{(k-1)}\) from \(A_{ij}\) 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 and there are no tasks in iteration \(k - 1\) reading value \(A_{kj}^{(k-1)}\) from \(A_{kj}\) either. Therefore, type II task \((k, j, k)\) with \(j \neq k\) is not anti-dependent on any tasks.

For a type III task \((k, i, j)\) with \(i \neq k \land j \neq k\), 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. \(i \neq k - 1 \land j \neq k - 1\). 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 \(k\) read \(A_{ij}^{(k-1)}\) from \(A_{ij}\).
2. \(i \neq k - 1 \land j = k - 1\). In this case, task \((k - 1, i, j)\) is of type II and no tasks in iteration \(k\) read \(A_{ij}^{(k-1)}\) from \(A_{ij}\).
A_{ij}^{(k-1)} from A_{ij}, either.

3. \( i = k - 1 \land j \neq k - 1 \). In this case, task \((k-1, i, j)\) is of type \(\Pi_i\) and the type \(\Pi_i\) tasks in \(\{(k-1, x, j) \mid x \neq k-1\}\) read \(A_{ij}^{(k-1)}\) from \(A_{ij}\). Therefore, task \((i, j)\) with \(i = k-1 \land j \neq k \land j \neq k-1 \land k > 0\) 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 4.

4. \( i = k - 1 \land j = k - 1 \). In this case, task \((k-1, i, j)\) is of type \(I_i\) and the type \(\Pi_i\) tasks in \(\{(k-1, x, j) \mid x \neq k-1\}\) and the type \(\Pi_{ij}\) 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 \(i = k-1 \land j = k-1 \land k > 0\) 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 4.

Similar to the flow data dependency, the anti data dependency successors of each task can be derived from the predecessors in Table 4. The Lemmas about anti data dependency successors for task \((k, i, j)\) is shown in Table 5. Lemmas anti-D2.1, anti-D2.2, anti-D3.1 and anti-D3.2 in Table 5 are the duels of Lemmas anti-2.1, anti-2.2, anti-3.1 and anti-3.2 in Table 4, respectively.

### 2.3 Output Data Dependency

In the tiled Gauss-Jordan elimination algorithm, each task \((k, i, j)\) overwrites \(A_{ij}\) written by task \((k-1, i, j)\). Recall that each task \((k, i, j)\) also reads \(A_{ij}\) written by task \((k-1, i, j)\). Thus, the output data dependency of the tiled Gauss-Jordan elimination algorithm is the same as flow data dependency specified in Lemmas flow-0 in Table 2 and flow-D0 in Table 3.

Because the output data dependencies are subsumed by the flow data dependencies in the Gauss-Jordan elimination, they need not be enforced.

### 3 Implementation in TBB

In TBB, 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 at the end of the task and spawn the successor if its ref_count reaches zero after the decrement.

To evaluate and execute the graph, the main driver simply spawns the tasks with no predecessors and wait for the completion of all the tasks without successors. The tasks will be executed according to the task graph, because a task will not be put in the task queue of the physical thread unless its ref_count becomes zero, and this happens only if all its predecessor tasks are completed.

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() of the task_graph class builds the task graph. It basically creates \(d^2\) tasks \((k, i, j)\) \((0 \leq k, i, j \leq d-1)\) referenced by m_tasks[k][i][j]. For each task, it sets the ref_count of the task to be the sum of the numbers of the flow and anti data dependency predecessors according to the Lemmas in Tables 2 and 4, respectively.

The computation of TBB task is specified in its execute() method. The algorithm of the execute() of the DagTask task in the implementation is shown in
execute() {
    if (i = k ∧ j = k) { //type I
        A_{ij} = (A_{ij})^{-1};
    } else if (i = k ∧ j ≠ k) { //type II
        A_{kj} = A_{kk} A_{kj};
    } else if (i ≠ k ∧ j = k) { //type II
        A_{ik} = -A_{ik} A_{kk};
    } else { //type III task with i ≠ k ∧ j ≠ k
        A_{ij} = A_{ij} - A_{ik} A_{kj};
    }
}

// decrement flow dependency successors
// Lemma flow-D0
if (k < d - 1) decspawn_set_zero(k + 1, i, j);
if (i = k ∧ j = k) {
    for y = 0 to d - 1 {
        if (y = k) continue;
        decspawn_set_zero(k, i, y)
    }
    for x = 0 to d - 1 {
        if (x = k) continue;
        decspawn_set_zero(k, x, j)
    }
} // Lemma flow-D2.1-2
if (i ≠ k ∧ j ≠ k) {
    for x = 0 to d - 1 {
        if (x = k) continue;
        decspawn_set_zero(k, x, j)
    } // Lemma flow-D3.1
    if (i ≠ k + 1 ∧ j = k + 1 ∧ k < d - 1) {
        for y = 0 to d - 1 {
            if (y = k + 1) continue;
            decspawn_set_zero(k + 1, i, y)
        } // Lemma flow-D3.3
        // decrement anti dependency successors
        if (i ≠ k ∧ j ≠ k)
            decspawn_set_zero(k, i, k); // Lemma anti-D2.1
        if (k < d - 1) {
            if (i ≠ k ∧ j = k + 1)
                decspawn_set_zero(k + 1, k, j); //Lemma anti-D2.2
            if (i ≠ k ∧ j ≠ k ∧ j ≠ k + 1)
                decspawn_set_zero(k + 1, k, j); //Lemma anti-D3.1
            if (i = k ∧ j = k ∧ i = k ∧ j ≠ k)
                decspawn_set_zero(k + 1, k, k); //Lemma anti-D3.2
        }
    }
}

Figure 2: Algorithm of execute() for task (k, i, j) in TBB

4 Overhead of CnC over TBB for Gauss-Jordan Algorithm

To measure the overhead of CnC over TBB, we run our TBB implementation and the CnC implementation [1] 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 problems sizes from 1024 to 10240 and the tile sizes from 8 to 256. We also run the sequential tiled Gauss-Jordan elimination in Figure 1 as the base sequential time in calculating speedups of parallel executions. For each configuration, we run the code for 5 times and take the average of the execution times as its execution time.

The best tile size we found is 64 for all the TBB, CnC and sequential tiled codes.

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

\[
R_{CnC/TBB} = \frac{e_{CnC}(n, t)}{e_{TBB}(n, t)}
\]

where \(e_{CnC}(n, t)\) and \(e_{TBB}(n, t)\) are the CnC and TBB execution times, respectively. Figure 3(a) shows the CnC/TBB time ratio, \(R_{CnC/TBB}\), for problem sizes from 1024 to 10240, using the optimal tile size 64. In particular, 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. Thus, the overhead of CnC over TBB is between 12% and 15% with tuner and between 18% and 22% without tuner.

We also calculated the GFLOPS (giga floating-point operations per second) performance by dividing the number of floating-point operations \(flop(n, t)\) by the execution time as follows:

\[
GFLOPS(n, t) = \frac{flop(n, t)}{10^9 \times e(n, t)}
\]
where \( flop(n, t) \) is as shown in (1) and \( e(n, t) \) is the execution time for problem size \( n \) and tile size \( t \). We plotted the GFLOPS of TBB, CnC with and without tuner, and the sequential tiled codes on 8 cores using the optimal tile size 64 for all the problem sizes from 1024 to 10240 in Figure 3(b).

![Figure 3: Performance of the Optimal Tile Size 64 Using 8 Cores](image)

We can also calculate \( \frac{GFLOPS_{CnC}(n, t)}{GFLOPS_{TBB}(n, t)} \) as the TBB/CnC ratio, which shows the percentage of the TBB performance that CnC can deliver (the TBB/CnC ratio is not plotted). Thus, the CnC/TBB time ratio between 1.15 and 1.12 tells that the CnC with tuner can deliver as much as 87% to 89% of the TBB performance. Likewise, the overhead between 18% and 22% of the CnC without tuner tells that the CnC without tuner delivers 82% to 85% of the TBB performance.

## 5 Conclusion

We make the following contributions in this paper: (1) We provide a complete data dependency analysis for the tiled in-place Gauss-Jordan elimination algorithm to guide the task-based parallelization. (2) We parallelized and implemented the tiled in-place Gauss-Jordan elimination in TBB. (3) 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% to 15% of the TBB time for the CnC with tuner, and 18% and 22% of the TBB time for the CnC without tuner.

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

## References


