# A Roofline-based Performance Estimator for Distributed Matrix-multiply on Intel CnC

Martin Kong, Louis-Noël Pouchet, P. Sadayappan
*Dept. of Computer Science and Engineering*
*The Ohio State University*
*Columbus, Ohio, USA*
{*kongm,pouchet,saday*}@*cse.ohio-state.edu*

***Abstract*—In this paper we show how to analytically model two widely used distributed matrix-multiply algorithms, Cannon's 2D and Johnson's 3D, implemented within the Intel Concurrent Collections framework for shared/distributed memory execution. Our precise analytical model proceeds by estimating the computation time and communication times, taking into account factors such as the block size, communication bandwidth, processor's peak performance, etc. It then applies a roofline-based approach to determine the running time based on communication/computation bottleneck estimation.**

**Our models are validated by comparing the estimations to the measured run times varying the problem size and work distribution, showing only marginal differences. We conclude by using our model to perform a predictive analysis on the impact of improving the computation speed by a factor of 4×.**

***Keywords*-Performance modeling, distributed computing, Intel Concurrent Collections**

## I. Introduction

Programs executing in clusters of shared-memory nodes are the *de facto* standard in parallel processing for scientific computing. Such clusters can be thought of in a simplified way as a 2-level parallel architecture, combining features of both distributed (top level) and shared memory machines (bottom level).

Numerous previous work addressed the problem of minimizing and approximating the modeling of communications in distributed environments [1], [2], [3], [4], [5]. Others focused on analyzing and comparing *pure MPI* and *hybrid MPI-OpenMP* execution models, e.g. [6], [7], [8], [9], [10]. However, little attention has been given to developing analytical models that put into play intra-node communication and inter-node communication on top of a run-time with the capability of executing in *pure MPI* fashion, but which automatically avoids communication using shared memory when the data is local to a node such as the Intel Concurrent Collections runtime [11]. Furthermore there is a compelling need for a model with good *predictive capability*, that is which enables the analytical exploration of different block sizes or processor frequencies without executing power consuming runs on a large-scale cluster.

In this work we focus on two widely used matrix-multiply algorithms, that we have efficiently implemented on top of the Intel Concurrent Collections (CnC) framework. We develop an analytical model that predicts their computation and communication times, considering both intra-node and inter-node communication. Using the roofline approach [1] we construct performance bounds taking into account machine-specific parameters (e.g., bandwidth, frequency, etc.) and application-specific ones (e.g., block size, communication expressions, etc.).

The rest of this paper is organized as follows. Section II introduces the related concepts and work relating to the implemented distributed algorithms and CnC; Section III briefly discusses how the algorithms are mapped to a CnC implementation; Section IV introduces the analytical model used to estimate the computation and communication time; Section V compares our analytical model to actual measurements; and Section VI uses the derived model to predict new performance bounds before concluding.

## II. Background

In this section we briefly recap the matrix-multiply algorithms implemented for this work. Also, we briefly summarize the main CnC entities and concepts. We assume that all matrices are of size $N \times N$, and that there are $p$ processors available.

### A. Cannon's 2D algorithm

Cannon's algorithm [2], [3], [4], [5] assumes a $p^{1/2} \times p^{1/2}$ grid of processors and is shown in Algorithm 1.

**Algorithm 1** Cannon's Algorithm

```
procedure CANNON($A$,$B$,$C$,$p$)
    for t $\in \{1 \ldots p^{1/2}\}$ do
        for all i,j $\in \{1 \ldots p^{1/2}\}$ do
            $C_{ij} \leftarrow C_{ij} + A_{ij} \cdot B_{ij}$
        end for
        Transfer $A_{ij}$ left-wise with wrap-around
        Transfer $B_{ij}$ up-wise with wrap-around
    end for
end procedure
```

The algorithm decomposes input matrices A and B into $(N/p^{1/2})^2$ blocks, distributing the blocks of matrix A such that processor $P_{i0}$ owns block $A_{ii}$, with the blocks wrapping around, i.e., $P_{ij}$ gets $A_{ij-i \mod p^{1/2}}$. A similar distribution is done for matrix B, but aligning blocks $B_{jj}$ to processor $P_{0j}$, i.e., $P_{ij}$ gets $B_{i-j \mod p^{1/2}j}$. Once the layout is set, Algorithm 1 performs $p^{1/2}$ block-matrix multiplications, followed by the appropriate block-shifts, left-wise for the blocks of matrix A, and up-wise for the blocks of matrix B.

## *B. Johnson's 3D algorithm*

Algorithm 2, Johnson's distributed matrix-multiply algorithm [12], [13], [14], [4], performs a 3D-parallel (along the i,j and k dimensions) multiplication, followed by a reduction along the k-dimension. Assuming that the distribution of matrix blocks is performed before the multiplication stage, this algorithm achieves the communication lower bound $O(N^2/p^{2/3})$, where the only communication required takes place during the reduction phase.

**Algorithm 2** Johnson's Algorithm

```
procedure JOHNSON($A$,$B$,$C$,$p$)
    for all i,j,k $\in \{1 \dots p^{1/3}\}$ do
        $P_{ij0}$ broadcasts $A_{ij}$ to all $P_{ijk}$
        $P_{0jk}$ broadcasts $B_{jk}$ to all $P_{ijk}$
        $C_{ijk} \leftarrow A_{ij} \cdot B_{jk}$
        $P_{ijk}$ contributes $C_{ijk}$ to a sum-reduction to $P_{i0k}$
    end for
end procedure
```

## *C. Concurrent Collections*

We implement Cannon's and Johnson's algorithm using the Intel CnC framework [15], [16]. Here we briefly describe the basics of a CnC program. A CnC program in Intel's framework is a C++ program using specific data structures to describe a program as (1) a set of computation steps; (2) the set of data items flowing between steps instances; and (3) tuning primitive to specify data and computation placement on physical entities [17].

*Tag Collections:* allow to identify individual instances of an object in a CnC program. For instance `(C : i)` denotes the $i^{th}$ dynamic instance of the object `C`. For the matrix multiplication case, tag collections typically takes the form of a triple {i,j,k}.

*Item Collections:* represent the sets of data elements manipulated by a CnC program. A CnC program can produce and consume several item collections. For instance, Cannon's implementation uses three item collections, one for each matrix, whereas Johnson's implementation relies on four, as shown later. In pure CnC, *programs implement the dynamic single assignment rule* for item collections, that is each tag value in a collection (e.g., `[A : i ]`) can be written only once, and read multiple times.

*Step Collections:* are the compute instances. They can consume and produce both items and tags. If a program has static control flow (e.g., matrix multiply), then all tags can be produced offline before starting the CnC graph execution. Dependence between step instances can be modeled using items and tags.

*CnC graph:* describes the relation between all the 3 types of entities above in a CnC program. A graph specifies the data that a computation consumes or produces, as well as the relations identifying related instances in the different collections.

*Data transfers:* CnC provides the very convenient accessors methods, **Get** and **Put**, which allow to receive and send data blocks (items). Each invocation of any of these methods should be accompanied by a **tag** instance that identifies the item. The DSA form is enforced by invoking a **Put** method with distinct tag values.

*Shared and distributed memory:* CnC implements a separation of concern between the *description* of the application as sets of tasks, data items, and producer/consumer relations between them; and *tuning annotations* that are specific to the execution context to enable higher performance for instance via explicit placement of tasks/data [17]. That is, from a high-level programmer's point of view, only Gets and Puts are used, irrespective of the executing environment: the same program can execute on a single processor, on a single node or on multiple nodes.

*CnC performance:* Previous work showed the effectiveness of CnC in delivering high performance on distributed environments on various applications including Cholesky decomposition and unbalanced tree search [17], on dense linear algebra kernels (e.g., Asynchronous Parallel Cholesky Factorization and Generalized Symmetric Eigensolver) outperforming ScaLAPACK+MPICH2/nemesis, multi-threaded MKL and equaling PLASMA+MKL [18], [19], while incurring in a low overhead w.r.t. Intel's TBB in single node execution [20]. This, in tandem with the CnC program semantics and its separation of concerns provide the ideal setup for high-performance tuning and space exploration.

## *D. The Roofline Model*

The Roofline Model [1] is a visual analytical model used to pinpoint performance bottlenecks. Figure 1 shows an example of the Roofline model plotting a machine with 16 GFLOPS of peak performance for an arbitrary program. The performance is bounded by (1) the communication speed, that is the time needed to bring in/out the data needed for the program; and by (2) the computation bound, that is the peak GF/s of the processor(s). When the operational intensity (i.e., the ratio of flops executed per byte read) increases beyond a certain value, the program performance becomes bounded by the computation peak.

The roofline idea has been extended for instance for specific applications and platforms [21], [22]; to model

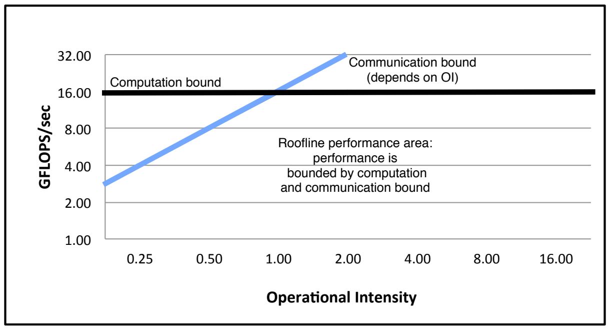

Figure 1. Roofline example

energy [23]; and to consider the implications of the cache hierarchy [24]. Few attempts have been made to adapt this model to consider intra-socket/node communication vs. inter-socket/node, as we do in the present work.

## III. Matrix Multiply Distributed-CnC Implementations

In this section we briefly describe Cannon's and Johnson's CnC implementations. We assume that all the data is initially at the node with rank-0, and that the matrix decomposition into data blocks is performed before the algorithm starts its execution. Furthermore: i) we do not assume that $p$ (the number of processors/ranks) is square or cube; ii) we add an additional program parameter, *block_size*, which defines the length of the block into which the matrices are decomposed; iii) for simplicity, we assume that block_size divides $N$, the matrix size. The values of block_size and $N$ determine the number of blocks into which each matrix is decomposed. We therefore can have more matrix blocks to compute than minimally required to use all available processors, giving us a degree of freedom in terms of parallelism implementation.

### A. Cannon's Implementation

Cannon's CnC implementation uses 3 item collections, one for each set of data blocks, one step collection and one tag collection. Each step instance is uniquely identified by a triple {i,j,k}. Furthermore, since CnC programs should be written in DSA form the items consumed and produced are identified by the step tag. Before initiating the actual computation, blocks of matrices A, B and C should be made available to the CnC environment via calls to the **Put** method with the proper tag value. Thus, the environment breaks matrices A, B and C into data blocks of size *block_size*, and puts blocks into the appropriate item collection (including blocks for C). Figure 2 shows the code that executes each Cannon step: retrieves a data-block from each matrix, performs the block matrix-multiply, and performs the data movement (left-wise for A, up-wise for B, and depth-wise for C).

```
i = tag.i;
j = tag.j;
k = tag.k;

ctx.mat_A_blocks.get(Triple(i,j,k), block_ik_A);
ctx.mat_B_blocks.get(Triple(i,j,k), block_kj_B);
ctx.mat_C_blocks.get(Triple(i,j,k), block_ij_C);

float * A = (*block_ik_A).addr();
float * B = (*block_kj_B).addr();
float * C = (*block_ij_C).addr();

mm_kernel (block_size, A, B, C);

int next_k = k + 1;
int next_i = i - 1;
int next_j = j - 1;

Triple nextA = SHIFTA(i,j,next_k,num_blocks);
Triple nextB = SHIFTB(i,j,next_k,num_blocks);
Triple nextC = SHIFTC(i,j,next_k,num_blocks);

ctx.mat_A_blocks.put(nextA, block_ik_A);
ctx.mat_B_blocks.put(nextB, block_kj_B);
ctx.mat_C_blocks.put(nextC, block_ij_C);
```

Figure 2. Cannon's CnC step

### B. Johnson's implementation

Johnson's implementation utilizes two step collections. The first one (see Figure 3) performs a 3D-parallel block matrix-multiply, while the second performs a reduction along dimension K. As in Cannon's case, each step instance of each collection is uniquely identified by a triple {i,j,k}. The environment produces all the data blocks of matrices A and B, together with the C-data block used during the reduction step (triples {i,j,0}). Notice that, unlike Cannon's step, Johnson performs only two **Get** operations since it will produce its own C-block product. Each {i,j,k} reduction step is 2D-parallel (see Figure 4), and consumes two C-blocks, the C-block produced by the block-multiply step identified by the triple {i,j,k} and the C-block reduced by the previous step of its own collection (reduction step {i,j,k-1}). The final blocks of matrix C, blocks {i,j,num_blocks} are then consumed by the environment and reassembled to produce matrix C. We note here that, depending on the MPI settings, the data blocks of matrices A and B can be assumed to have been distributed before the actual matrix-multiply steps initiate execution. For guaranteeing this, we set the I_MPI_INTRANODE_EAGER_THRESHOLD variable to a size bigger than the largest message. For the purpose of our experiments we set this value to 20MB.

Johnson's reduction steps in Figure 4 can be executed as soon as both the previously reduced C-block and a new $C_{ijk}$ block from Figure 3 becomes available, i.e., a data-dependence. Regarding the DSA form, we observe that each step instance of Figure 3 produces a fresh $C_{ijk}$ block, uniquely identified by its {i,j,k} tag value. Similarly, in the

```
i = tag.i;
j = tag.j;
k = tag.k;

ctx.mat_A_blocks.get(Triple(i,k,j), block_ik_A);
ctx.mat_B_blocks.get(Triple(k,j,i), block_kj_B);

float * A = (*block_ik_A).addr();
float * B = (*block_kj_B).addr();
float * C = (*block_ijk_C).addr();

mm_kernel (block_size, A, B, C);

ctx.mat_C_blocks.put(Triple(i,j,k), block_ijk_C);
```

Figure 3. Johnson's CnC block-multiply step

reduction step, these blocks are uniquely identified by the same tag and are kept in sync. Each newly reduced C-block is associated to the next value along the k-dimension, which functions as a time-dimension.

```
i = tag.i;
j = tag.j;
k = tag.k;

ctx.mat_C_blocks.get(tag, block_ijk_C_in);
ctx.mat_C_red_block.get(tag, block_ij_C_out);

float * Cin = (*block_ijk_C_in).addr();
float * Cout = (*block_ij_C_out).addr();

mm_reduction (block_size, Cin, Cout);

int next_k = k + 1;
ctx.mat_C_red_block.put
  (Triple(i,j,next_k), block_ij_C_out);
```

Figure 4. Johnson's CnC reduce-step

### C. Step kernel

The matrix multiplication between blocks used both implementations is performed by a single kernel function we implemented, which is tiled for L3 (12 MB per socket) and L1 cache (32 KB per core), unrolled by 4× along the k-dimension of the L1 tile, and decorated with SIMD and vector pragmas. The L3 tile sizes used throughout the whole set of experiments were $\{240 \times 256 \times 128\}$, while for the L1 tiles $\{80 \times 128 \times 16\}$ were used.

## IV. Analytical Model

In this section we introduce an analytical model to estimate the execution time of distributed-CnC implementations. The model predicts the computation and communication time for the previously introduced CnC matrix-multiply implementations. We first describe the general model which is algorithmic independent, and then proceed to tailor it to each individual case.

### A. Base Model

Table I shows the input parameters to our general model. For simplicity, we assume that both algorithms handle square matrices, use single precision (4 bytes per value), that there is no over-subscription of ranks to cores, and that each MPI process uses a single core. Despite these assumptions, the model can be easily extended to other scenarios. We separate the parameters into two sets, the problem dependent ones (top) and the platform/network specific ones (bottom).

| Parameter Name | Description |
|---|---|
| $N$ | Problem size (along one dimension) |
| $B$ | Block size (along one dimension) |
| $M$ | Number of nodes |
| $P$ | Number of MPI processes/ranks |
| $BW_{intra}$ | Intra-node bandwidth (MB/s) |
| $BW_{inter}$ | Inter-node bandwidth (MB/s) |
| $c_{node}$ | Cores per node |

Table I
Model input parameters

Table II shows the derived variables used in the general model. The top set of metrics relate to the problem decomposition, the middle set are performance metrics, and the bottom ones communication variables. These metrics are still used in an algorithm independent fashion.

| Metric Name | Description |
|---|---|
| $b$ | Number of data blocks per dimension (algorithm independent) |
| $steps$ | Number of steps (algorithm independent) |
| $msgs$ | Number of messages per step (algorithm dependent) |
| $F_{par}$ | Parallel factor (algorithm dependent) |
| $F_{ser}$ | Serial factor (algorithm dependent) |
| $W_{intra}$ | Intra-socket work fraction (algorithm dependent) |
| $W_{inter}$ | Inter-socket work fraction (algorithm dependent) |
| $T$ | Program time |
| $T_{comp}$ | Compute time |
| $T_{comm}$ | Communication time |
| $T_{step}$ | Step compute time |
| $T_{intra}$ | Intra-socket communication time |
| $T_{inter}$ | Inter-socket communication time |
| $L$ | Message (data block) length in MB |
| $V$ | Total communication volume (MB) |
| $V_{intra}$ | Intra-socket communication volume (MB) |
| $V_{inter}$ | Inter-socket communication volume (MB) |

Table II
General performance derived metrics

Equations (1) and (2) show how the problem decomposition variables are derived. In general, these are problem specific but algorithm independent. Each matrix is decomposed into $b^2$ blocks of size $B^2$. The number of compute steps is a function of $b$. We note that although we have $steps$ to execute, the order in which these are computed are defined by the algorithm properties as well as the CnC implementation.

$$b = N/B \tag{1}$$

$$steps = b^3 \tag{2}$$

Our model follows the common computation and communication split, while also taking into account if communication can be restricted to within a socket/node, or if it requires a remote access. The total compute time is estimated by the product of the number of steps executed by a task and the projected kernel time of the step $T_{step}$. The formulas used for estimating each of the performance variables are listed in Equations (3)-(8), using terms defined later on.

$$T = T_{comp} + T_{comm} \tag{3}$$

$$T_{comp} = F_{ser} \times T_{step} \tag{4}$$

$$T_{step} = \text{measured on the test machine} \tag{5}$$

$$T_{comm} = T_{intra} + T_{inter} \tag{6}$$

$$T_{intra} = V_{intra}/BW_{intra} \tag{7}$$

$$T_{inter} = V_{inter}/BW_{inter} \tag{8}$$

As $T_{step}$ is the time to execute one single step on a single core, it is very practical to measure this time on the target machine. For simplicity purpose we assume $T_{step}$ does not vary with the execution context (block size, etc.) which, as we show later on, is a valid approach in our experimental setup. The general problem of predicting DGEMM kernel performance analytically (e.g., [25]) is a difficult task, and out of the scope of this paper which focuses on proper modeling of communication times.

The communication volume is broken down into local (intra-node) and remote (inter-node) communication. Both volumes are obtained from the total communication volume ($V$), by applying the respective work factor ($W_{intra}$ or $W_{inter}$). These factors are highly dependent of the algorithm and of the node and core distribution (e.g., block, cyclic, etc). In particular, we assume that the mapping of ranks to nodes and cores follows a block distribution. The total communication volume is the product of the number of steps executed by a rank ($F_{ser}$), the message size ($L$) and the number of messages per step ($msgs$). The reason to include this factor is to model accesses of different ranks sharing a network device. Equations (9)-(12) show how we derive the communication related variables, using terms defined later on in an algorithmic dependent way. We remark that for utmost precision one can refine this model by considering different bandwidth for local communications between processes mapped to cores on the same socket versus on different sockets, however for simplicity purpose we assume a single intra-node bandwidth here.

$$V_{intra} = V \times W_{intra} \tag{9}$$

$$V_{inter} = V \times W_{inter} \tag{10}$$

$$V = F_{ser} \times L \times msgs \times F_{overhead} \tag{11}$$

$$L = 4B^2/2^{20} \tag{12}$$

## B. Cannon Model Specifics

We now define the model variables specific to Cannon's algorithm. This algorithm has $O(b^2)$ parallelism and executes $b$ stages. Each step performs three **Get** operations. Hence $msgs = 3$. The factor $W_{intra}$ can be interpreted as follows: if only one node is used, then the communication volume is affected by a factor equal to $P \times \mathcal{C}$, where $\mathcal{C}$ is a contention factor representing the number of cores accessing the shared memory. Otherwise, the contention will be for the network device, and the factor becomes the number of cores per node. The additional $1/3$ factor comes from only having to access the blocks of matrix B from another node. C-blocks will always be local, and $C_{node} - 1$ cores will access an A-block from the same node (due to the left-wise shift). For not over complicating the formula, we leave it as 1 for the A-blocks. Equations (13)-(18) define these metrics. This assumption is only valid when the distribution of processes to cores is done in a block fashion.

$$msgs = 3 \tag{13}$$

$$F_{overhead} = 1.5 \tag{14}$$

$$F_{par} = \min(P, b^2) \tag{15}$$

$$F_{ser} = \lceil steps/F_{par} \rceil \tag{16}$$

$$W_{intra} = \min(P, c_{node}) \times (\min(P, c_{node}) - 1) \tag{17}$$

$$W_{inter} = if(P > c_{node}) \;\; then \;\; \frac{c_{node}}{3} \;\; else \;\; 0 \tag{18}$$

## C. Johnson Model Specifics

We now proceed to define the specific variables of the Johnson CnC implementation. It consists of two step collections: the $O(b^3)$-parallel block matrix multiply, and the $O(b^2)$-parallel block reduction step, performed along dimension k. We note that for simplicity purposes, the compute time of the reduction step is ignored due to its lower complexity w.r.t. the block matrix-multiply step ($O(B^2)$ vs. $O(B^3)$ computations). Thus, the compute time of both models will be identical for the same set of parameters. Both step collections consist of $steps$ instances. The $\min$ function models the possibility of having fewer blocks than processes along a particular dimension. We only model the communication happening in the reduction stage, since we assumed that the block matrix-multiply step already had received all the required input blocks before they start execution. The communication distributing factors, $W_{intra}$ and $W_{inter}$, essentially say that if the communication is happening in a single node, then the $W_{intra}$ of the Cannon

CnC implementation applies, otherwise the original communication volume is halved, since the previously reduced C-block will always remain in the same step. Equations (19)-(24) define the formulae used to derive these metrics.

$$msgs = 2 \tag{19}$$

$$F_{overhead} = 1 \tag{20}$$

$$F_{par} = \min(P, b^3) \tag{21}$$

$$F_{ser} = \lceil steps/F_{par} \rceil \tag{22}$$

$$W_{intra} = \min(P, c_{node}) \times (\min(P, c_{node}) - 1) \tag{23}$$

$$W_{inter} = if(P > c_{node}) \;\; then \;\; \frac{c_{node}}{2} \;\; else \;\; 0 \tag{24}$$

## V. Model Validation

In this section we validate our analytical model by comparing the estimated compute time and communication time to the measured values the Cannon and Johnson CnC implementations.

| Parameters | Value |
|---|---|
| Nodes | 4 |
| Processor | Intel Xeon E5630 @ 2.5 GHz |
| Sockets per node | 2 |
| Cores per socket | 4 |
| Intra-node bandwidth | 25000 MB/s |
| Inter-node bandwidth | 1250 MB/s |
| L1 Cache | 32 KB per core |
| L2 Cache | 256 KB per core |
| L3 Cache | 12 MB per socket |
| CnC | 1.01 |
| MPI run-time | Intel MPI 5.0 |
| Compiler | ICPC 13 |

Table III
Experimental setup

| Software | Enabled options |
|---|---|
| ICPC | -xhost -O3 –fno-alias -std=c99 |
| CNC | DIST_CNC=MPI CNC_NUM_THREADS=1 CNC_MPI_SPAWN=T |
| I_MPI | I_MPI_EAGER_THRESHOLD=20MB I_MPI_SHM_BYPASS=1 I_MPI_INTRANODE_EAGER_THRESHOLD=20MB I_MPI_DAPL_TRANSLATION_CACHE=0 |
| Slurm | –nodes=M –ntasks=T –ntasks-per-socket=$\min(T, 4)$ –threads-per-core=1 –distribution=block –contiguous |

Table IV
Software stack flags

| Configuration | M | P | Sockets |
|---|---|---|---|
| map1 | 1 | 1 | 1 |
| map4 | 1 | 4 | 1 |
| map8 | 1 | 8 | 2 |
| map16 | 2 | 16 | 4 |
| map32 | 4 | 32 | 8 |

Table V
Evaluated task mappings

### A. Experimental setup

Table III summarizes the hardware setup and the software stack used to evaluate the two CnC implementations and to validate our analytical model.

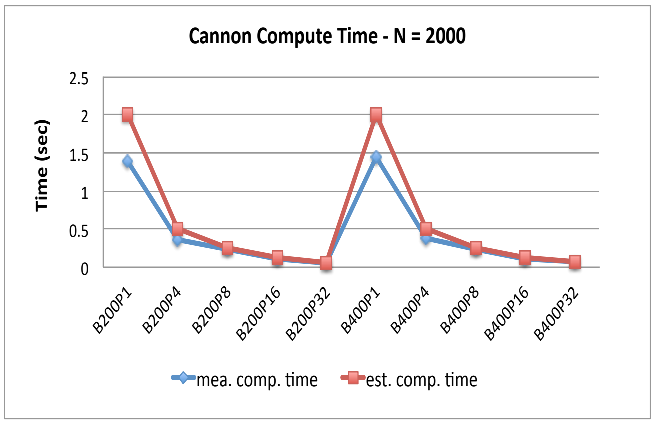

Figure 5. Cannon compute time for N=2000

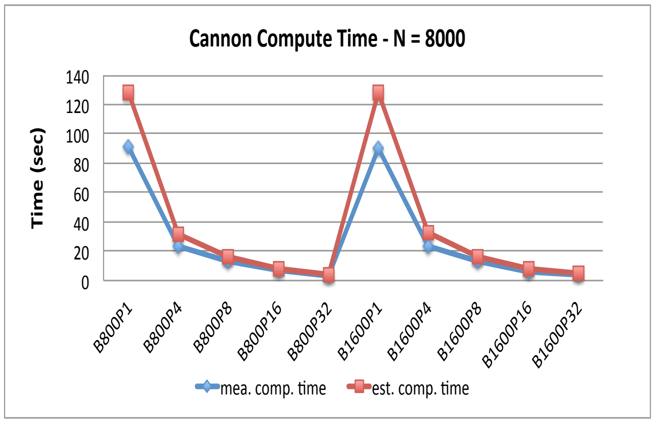

Figure 6. Cannon compute time for N=8000

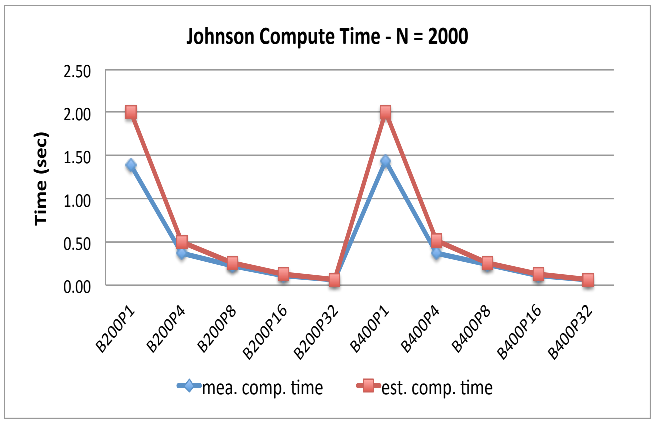

Figure 7. Johnson compute time for N=2000

Table IV summarizes the non-default flags and options used for each layer of the software stack. Of special importance are the I_MPI flags that affect the choice of eager vs. rendezvous transfer protocol, and the disabling of the DAPL translation cache. The former enables the run-time to send messages as soon as they become available rather

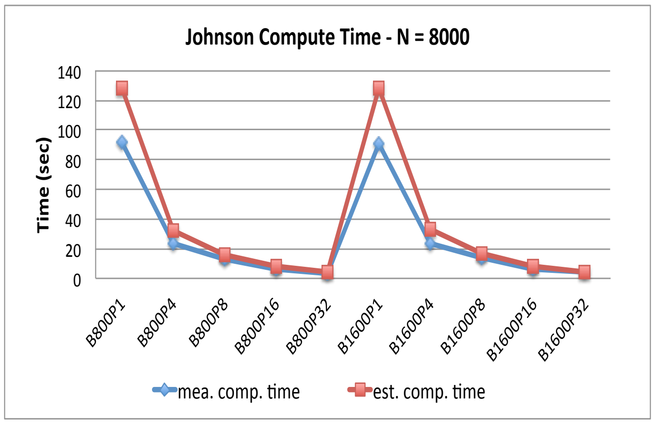

Figure 8. Johnson compute time for N=8000

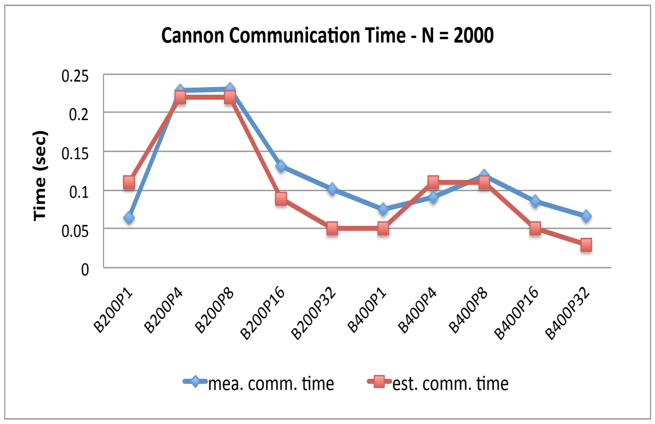

Figure 9. Cannon communication time for N=2000

than waiting for the time at which a process requires them, i.e., these flags enable the run-time to favor a *push model* over a *pull model*. The relevance of these options is that they help in making more precise time estimations, e.g., for the Johnson implementation, where the input matrices have to be replicated and distributed, a program stage which we do not account in our experiments and modeling. Finally, the DAPL cache is disabled to avoid potential correctness errors.

For each CnC implementation we evaluate two problem sizes (N={2000,8000}), 2 block sizes (B={N/10,N/5}) and five task mappings. The task mappings are listed in Table V. Lastly, throughout this section all x-labels follow the format "B[block-size]P[num-cores]".

## B. Model fitting

We validate the accuracy of our model by comparing the estimated compute and communication time to the measured *compute-only* and *communication-only* execution times. The compute-only time is obtained by timing the kernel of each step instance, adding and then averaging by the number of steps executed by each process. The communication-only time is measured when executing the original CnC program without invoking the kernel code.

Figures 5, 6, 7, and 8 show the fitting of the compute estimation time for both CnC programs. We observe that the estimated curve better matches the big problem size for both algorithms. In addition, the estimated time when using a single core is over-estimated since we do not take into account temporal data locality effects (e.g., reuse of data tiles between step instances) when the L3 does not need to be shared between processors. The curves on both implementations are identical for the same problem size because we decided to ignore the compute time of Johnson's reduction step (it performs $O(n^2)$ operations).

The following four figures (9 - 12) show how the estimated communication time matches the measured time. Overall we see that the model estimations follow closer the big problem size than the small one. The reason for this is that the data block sizes (parameter B) used are for N=8000 produce bigger messages than for N=2000. This optimizes the payload-to-overhead ratio.

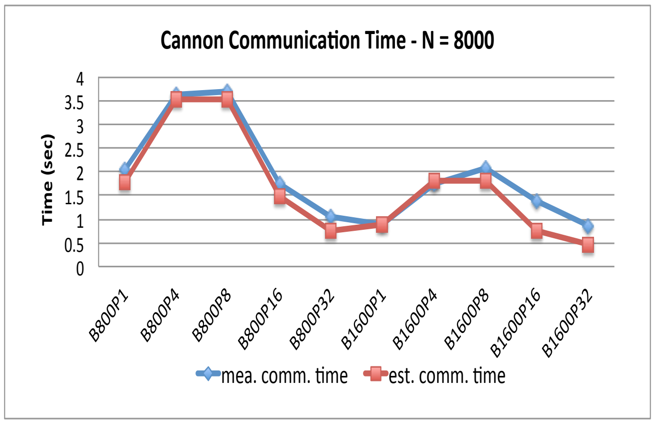

Figure 10. Cannon communication time for N=8000

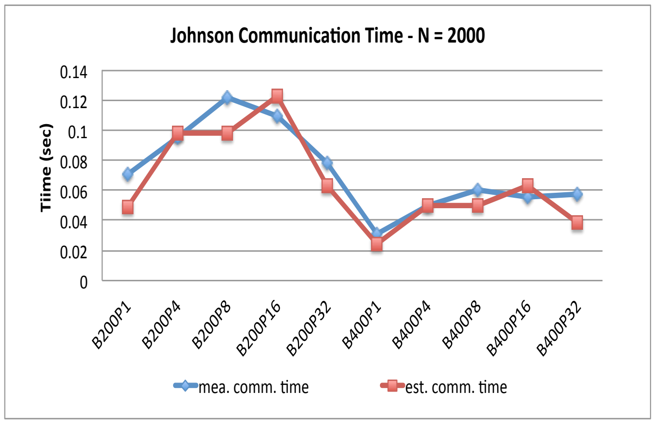

Figure 11. Johnson communication time for N=2000

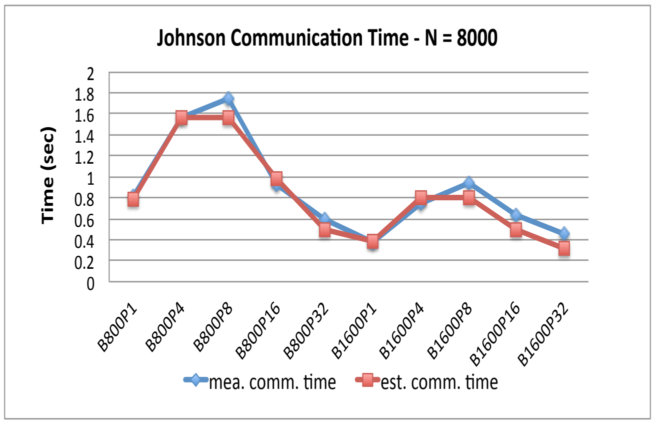

Figure 12. Johnson communication time for N=8000

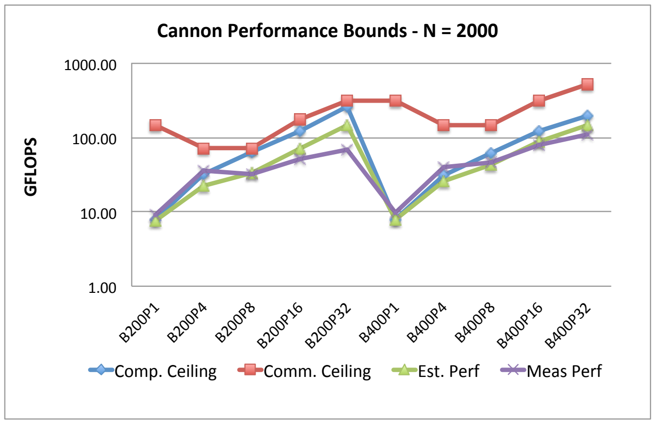

Figure 13. Cannon perfbound model for N=2000

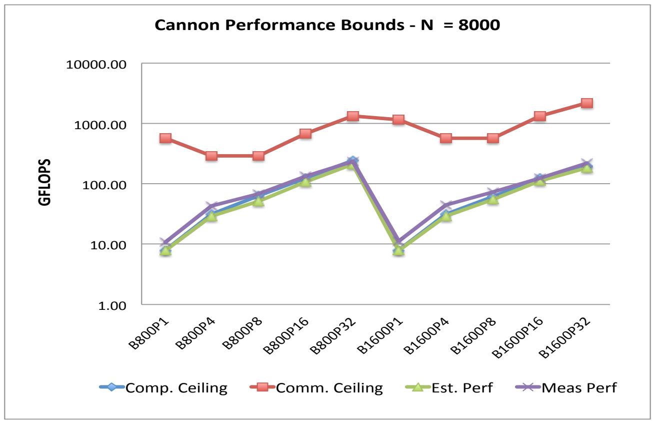

Figure 14. Cannon perfbound model for N=8000

## C. Compute and Bandwidth Bounds

We now use the previously estimated compute and communication times to observe whether we are compute or bandwidth bound. We plot both the estimated compute bandwidth bound for every work-decomposition and task-mapping configuration (i.e., the N, B, M and P parameters),

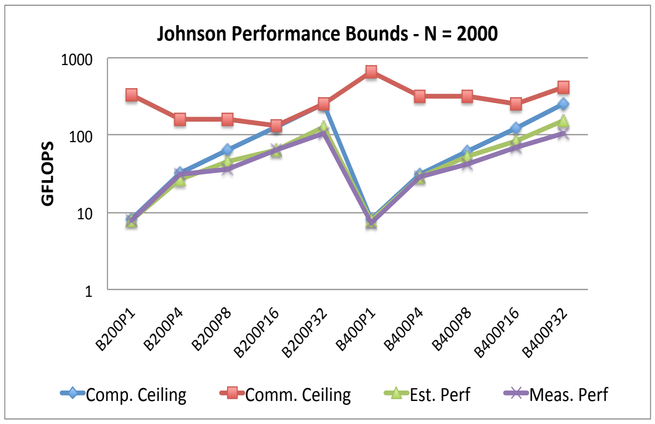

Figure 15. Johnson perfbound model for N=2000

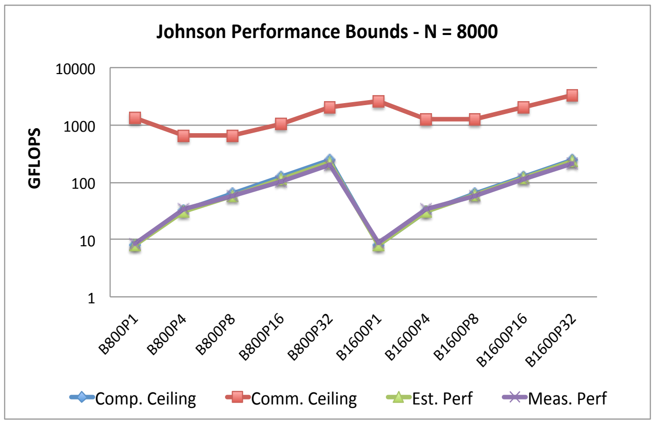

Figure 16. Johnson perfbound model for N=8000

since these define the compute time and the bandwidth requirements. We also plot the estimated final performance ($T = T_{comp} + T_{comm}$ from our analytical model) and the measured final performance. Figures 13 to 16 show how the performance (GFLOPS) varies as one adds more compute resources.

Figure 13 and Figure 15 show Cannon's and Johnson's performance bounds for N=2000, while Figure 14 and Figure 16 show the model for N=8000. One can observe that for $\{N,B,P\}=\{2000,200,16\text{-}32\}$ both algorithms are close to be bandwidth bound. This happens because the ratio of operation per inter-node communication for B=200 is $100 = 200^3 * 2/(200^2 * 4)$ as only one matrix block needs to be communicated inter-node. In addition, the inter-node bandwidth of 1250MB/s is shared between 8 cores. Thus the effective bandwidth is 156.25MB/s, leading to a bound of 16 GFLOPS per core. The theoretical machine peak assuming a vector width of 4 is about 19.2 GF/s. The estimated performance for these configurations is between 60-70 GFLOPS on 16 cores and 130-140 on 32 cores, yielding approximatively 4.5 GFLOPS per core. This performance

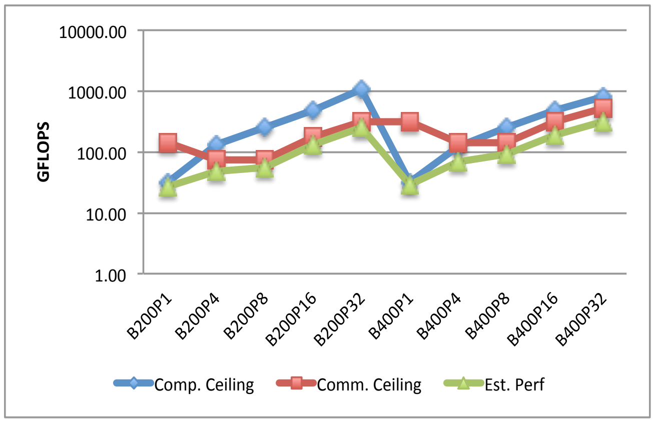

Figure 17. Cannon performance bound model for N=2000 assuming a 4x step kernel speedup

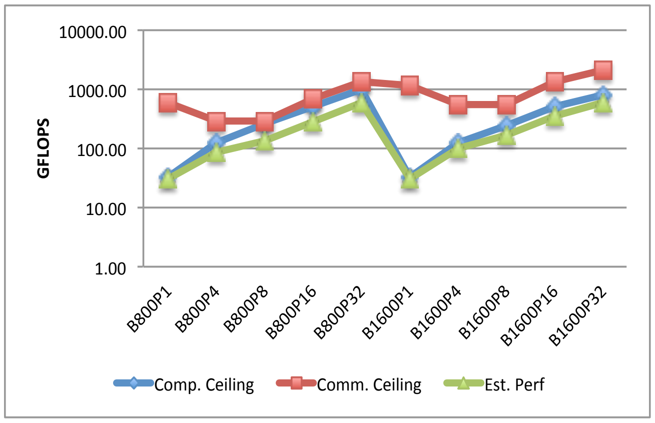

Figure 18. Cannon performance bound model for N=8000 assuming a 4x step kernel speedup

can be increased by improving the kernel computation time, however our analysis shows it would not be useful (in terms of execution time) to improve the kernel performance beyond 16 GF/s single-core, unless incrementing the block size.

A similar reasoning can be employed to understand the performance bounds of both algorithms for N=8000 on Figures 14 and 16.

## VI. Exploratory Analysis

In this section we perform an analytical exploration to see how the performance bound of both CnC implementations vary when the kernel step would be optimized to achieve a $4\times$ faster execution time compared to our actual implementation used V-C. In other words, we use our analytical model to determine whether it would be worth spending additional tuning time on the elementary block matrix-multiply operation, and what could be the expected gains.

Figures 17-18 show that such improvement in the kernel step time will cause the Cannon CnC implementation to be mostly bandwidth bound for N=2000, with the exceptions of P=1, and close to be bandwidth bound for N=8000 on P=16 and P=32. Regarding Johnson's implementation, Figures 19-20 show that improving the computation bound by a factor of 4 narrows the gap w.r.t. the bandwidth curve. This program is still compute bound for N=8000, and partially bandwidth bound for N=2000, as previously discussed.

## VII. Conclusion and Future Work

In this paper we showed how to implement two widely used algorithms for distributed matrix multiplication, Cannon and Johnson, on top of the Intel Concurrent Collections runtime. Based on CnC concepts we designed an analytical model to estimate both the computation and communication time, considering several machine-specifics and algorithm properties and metrics. Emphasis was put on estimating the communication time, since this is often the limiting

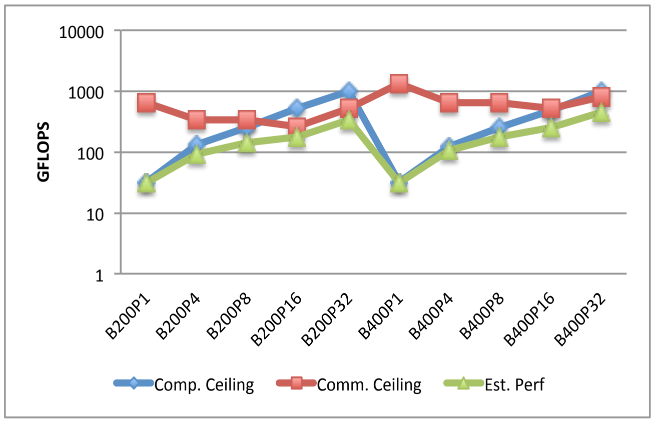

Figure 19. Johnson performance-bound model for N=2000 assuming a 4x step kernel speedup

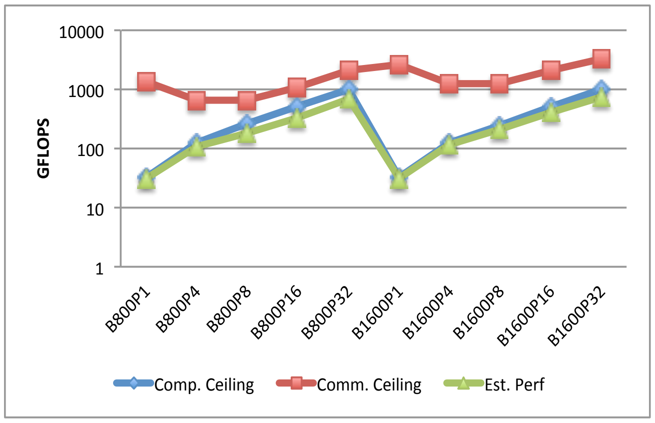

Figure 20. Johnson performance-bound model for N=8000 assuming a 4x step kernel speedup

resource in distributed computing. We have experimentally validated our model, and conducted a predictive study on the performance of Johnson and Cannon algorithms reachable on our distributed computing setup.

As future work we will consider refining the computation model, and extending this approach to other classical distributed computing algorithms, including recent developments on 2.5D algorithms.

*Acknowledgments:* This work was supported in part by the U.S. National Science Foundation through grants 0926127 and 1321147, and by the U.S. Department of Energy through grant DE-SC0008844.

## References

[1] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[2] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5 d matrix multiplication and lu factorization algorithms," in *Euro-Par 2011 Parallel Processing*. Springer, 2011, pp. 90–109.

[3] L. E. Cannon, "A cellular computer to implement the kalman filter algorithm." DTIC Document, Tech. Rep., 1969.

[4] G. H. Golub and C. F. Van Loan, *Matrix computations*. JHU Press, 2012, vol. 3.

[5] H.-J. Lee, J. P. Robertson, and J. A. Fortes, "Generalized cannon's algorithm for parallel matrix multiplication," in *Proceedings of the 11th international conference on Supercomputing*. ACM, 1997, pp. 44–51.

[6] L. Adhianto and B. Chapman, "Performance modeling of communication and computation in hybrid mpi and openmp applications," *Simulation Modelling Practice and Theory*, vol. 15, no. 4, pp. 481–491, 2007.

[7] N. Drosinos and N. Koziris, "Performance comparison of pure mpi vs hybrid mpi-openmp parallelization models on smp clusters," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. IEEE, 2004, p. 15.

[8] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes," in *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*. IEEE, 2009, pp. 427–436.

[9] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha, "A framework for performance modeling and prediction," in *Supercomputing, ACM/IEEE 2002 Conference*. IEEE, 2002, pp. 21–21.

[10] D. S. Henty, "Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling," in *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2000, p. 10.

[11] Intel, "Intel concurrent collections for c++." [Online]. Available: https://icnc.github.io

[12] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar, "A three-dimensional approach to parallel matrix multiplication," *IBM Journal of Research and Development*, vol. 39, no. 5, pp. 575–582, 1995.

[13] S. L. Johnsson, "Minimizing the communication time for matrix multiplication on multiprocessors," *Parallel Computing*, vol. 19, no. 11, pp. 1235–1257, 1993.

[14] E. Dekel, D. Nassimi, and S. Sahni, "Parallel matrix and graph algorithms," *SIAM Journal on computing*, vol. 10, no. 4, pp. 657–675, 1981.

[15] Z. Budimlic, A. Chandramowlishwaran, K. Knobe, G. Lowney, V. Sarkar, and L. Treggiari, "Multi-core implementations of the concurrent collections programming model," in *CPC?09: 14th International Workshop on Compilers for Parallel Computers*, 2009.

[16] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach *et al.*, "Concurrent collections," *Scientific Programming*, vol. 18, no. 3, pp. 203–217, 2010.

[17] F. Schlimbach, J. C. Brodman, and K. Knobe, "Concurrent collections on distributed memory theory put into practice," in *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*. IEEE, 2013, pp. 225–232.

[18] A. Chandramowlishwaran, K. Knobe, and R. Vuduc, "Performance evaluation of concurrent collections on high-performance multicore computing systems," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–12.

[19] P. Tang, "Measuring the overhead of intel c++ concurrent collections over threading building blocks for gauss–jordan elimination," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 18, pp. 2282–2301, 2012.

[20] A. Chandramowlishwaran, K. Knobe, and R. Vuduc, "Applying the concurrent collections programming model to asynchronous parallel dense linear algebra," in *ACM Sigplan Notices*, vol. 45, no. 5. ACM, 2010, pp. 345–346.

[21] K.-H. Kim, K. Kim, and Q.-H. Park, "Performance analysis and optimization of three-dimensional fdtd on gpu using roofline model," *Computer Physics Communications*, vol. 182, no. 6, pp. 1201–1207, 2011.

[22] T. Cramer, D. Schmidl, M. Klemm, and D. an Mey, "Openmp programming on intel r xeon phi tm coprocessors: An early performance comparison," 2012.

[23] J. W. Choi, D. Bedard, R. Fowler, and R. Vuduc, "A roofline model of energy," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 2013, pp. 661–672.

[24] A. Ilic, F. Pratas, and L. Sousa, "Cache-aware roofline model: Upgrading the loft," *Computer Architecture Letters*, vol. 13, no. 1, pp. 21–24, Jan 2014.

[25] E. Peise and P. Bientinesi, "Performance modeling for dense linear algebra," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, Nov 2012, pp. 406–416.