ASTES

Advances in Science, Technology and Engineering Systems Journal
Vol. 5, No. 4, 824-840 (2020)
www.astesj.com
Special Issue on Multidisciplinary Innovation in Engineering Science & Technology

ASTES Journal
ISSN: 2415-6698

Fine Tuning the Performance of Parallel Codes

Sanaz Gheibi*, Tania Banerjee, Sanjay Ranka, Sartaj Sahni

Department of Computer and Information Science and Engineering, University of Florida, Gainesville, 32611, USA

ARTICLE INFO

Article history:
Received: 03 June, 2020
Accepted: 11 August, 2020
Online: 31 August, 2020

Keywords:
Parallel speed up
Large matrices
Performance fine tuning

ABSTRACT

We propose a multilevel method to speed highly optimized parallel codes whose runtime increases faster than their workload. This method requires the ability to solve large instances by decomposing them into smaller instances. Using a simple parallel computing model, we derive a mathematical model that predicts whether or not our method can improve performance and also predicts the amount of improvement attainable. Our method is tested and shown to be effective on three highly optimized BLAS (Basic Linear Algebra Subprograms) routines from Intel's Math Kernel Library (MKL). Those routines are cblas_dgemm, cblas_dtrmm and cblas_dsymm. On the Intel Knights Landing (KNL) platform our method speeds cblas_dgemm by 33%, cblas_dtrmm by 50% and cblas_dsymm by 49% on double-precision matrices of size $16K \times 16K$*, when the KNL's default memory-clustering configuration (cache-quadrant) is used.*

1 Introduction

This paper is an extension of our previous paper on fine tuning a group of linear algebra libraries using a multi-level approach [1].

Our problems of interest are a group of parallel codes whose runtime grows faster than the problem size. The following example provides a quick demonstration of how such parallel codes behave and what motivates our proposed method.

Table 1 contains the runtime values for multiplying square matrices of different sizes on the intel KNL multi-level memory multicore computer. The memory-clustering configuration is set to its default which is cache quadrant. The multiplications are done in double precision using the highly optimized cblas_dgemm function from Intel MKL. The runtimes are obtained using either 32 or 64 cores. For each set of cores and each matrix size of $2N \times 2N$, the table also provides the runtime ratio when the matrix size increases from $N \times N$ to $2N \times 2N$.

Since the matrix dimensions are increasing by a factor of 2 (thus increasing the workload by a factor of 8), we expect all the ratios to be 8. However, that is not the case and four of the ratios are greater than 8. That is where the runtime increases faster than the workload and that brings up possibilities of improvement. Here, for both 32 cores and 64 cores, we get better results if instead of performing a $16K \times 16K$ ($64K \times 64K$) multiplication, we perform 8 instances of $8K \times 8K$ ($32K \times 32K$) multiplications and combine the results (each input matrix is partitioned into 4 blocks of half the dimension; the result is obtained by doing 8 block multiplies and 4 block additions). The previous statement is valid as long as the overhead of performing 4 pairs of matrix additions is less than the runtime reduction obtained by multiplying 8 smaller instances.

We can further reduce the runtime by running the smaller instances in parallel. For example, instead of running one instance of $32K \times 32K$ multiplication using 64 cores, we can run 4 parallel pairs of $16K \times 16K$ multiplications using 32 cores in less total time. Again, we will have speed up if the overhead of adding 4 pairs of matrices is less than the obtained runtime reduction.

Table 1: Run times of double-precision matrix multiplications using 32 and 64 cores. Ratio is obtained by dividing the run time in the current row by the runtime in the previous row.

Matrix Size	32 Cores		64 Cores	
	Time(seconds)	Ratio	Time(seconds)	Ratio
$4K \times 4K$	0.37	-	0.32	-
$8K \times 8K$	2.32	6.27	1.51	4.71
$16K \times 16K$	20.56	8.86	12.32	8.16
$32K \times 32K$	159.01	7.73	97.52	7.92
$64K \times 64K$	1517.21	9.64	871.10	8.93

Our proposed method tries to improve the speed of the parallel codes whose runtime increases faster than the workload. This method uses multiple levels and works on applications that are of a decomposable nature. We start with a 2-level algorithm. At level 2, the problem is broken into a number of subproblems that are solved (serially or in parallel) using the 1-level algorithm and then

*Corresponding Author: Sanaz Gheibi, Dept. of Computer and Information Science and Engineering, University of Florida, Gainesville, Florida, USA, sgheibi@ufl.edu

the results are combined to form the final solution. (By 1-level algorithm we mean the parallel and highly efficient algorithm whose runtime grows faster than its workload). Now, suppose the 2-level algorithm itself demonstrates the property that its runtime grows faster than its workload. In this case, we can gain further speed up by using a 3-level algorithm in which the problem is broken into a proper number of subproblems which are solved using the 2-level algorithm. We can continue increasing the number of levels in a similar fashion.

While our proposed method has much in common with divide and conquer, block matrix multiplication and Cannon's method for matrix multiplication [2], there are subtle differences. In a typical divide and conquer algorithm [3]- [7] the division scheme is the same across the levels (an example being the Strassen's matrix multiplication algorithm [3]); whereas in our method the number of subproblems into which the problem is broken can be different in different levels. Block matrix multiplication has been used both in serial and parallel matrix multiplication algorithms [2, 8, 9]. It can be viewed as a 1-level divide-and-conquer algorithm; while our method uses multiple levels of decomposition and combination. Our method is different from Cannon's algorithm in both the number of levels and the block's movement scheme. Cannon's movement scheme is designed for mesh-connected parallel computers with wraparound and uses row-and-column circular shifts. Our method uses a serpentine movement scheme that will be described in section 3.2.

Regardless of the above-mentioned similarities, the value of our method is that it can speed up codes whose runtimes grow faster than their workload. Here, the codes to be optimized are viewed as black boxes that can be already parallel, highly optimized and have their own complexities. As a result, this method can be applied to a range of problems broader than just linear algebra libraries.

We use a simple parallel computing model to formulate the conditions when our method results in speed up and also predict the amount of speed up. In section 5 we will demonstrate the effectiveness of our algorithm and our prediction formula on three linear algebra libraries from Intel MKL.

Our experimental results are obtained using the Knights Landing (KNL) computer. Although this architecture has been discontinued by Intel, we believe our fine-tuning method remains relevant for the following reasons. First, Knights Landing clusters are still used in many national laboratories and second, our method can be used in future architectures that have multilevel memory.

The rest of the paper is organized as follows. Section 2 summarizes some of the related work. In Section 3 we describe a solution framework for general problems and then describe its specific application to a class of linear algebra functions. Section 4 gives a brief description of KNL architecture. Section 5 presents the experimental results; and we conclude in Section 6.

2 Related Work

This section summarizes some of the previous methods used to speed up applications.

Tiling has been used extensively to improve data locality and cache efficiency. This method works by breaking the problem into subproblems or tiles in such a way that the amount of memory reuse in faster levels of memory hierarchy increases. Tiling has been used in a large group of problems including sparse and dense matrix-matrix multiplication [10, 11], parallel tensor transpose [12] and LU factorization [11]. In [13] the authors use tiling along with thread batching and architecture specific optimizations to speed up Alternating Least Squares algorithm used in recommender systems. They use a fine grained tiling in order to mitigate the imbalance in threads' workload that results from the sparse nature of the problem at hand. In [14] the authors use temporal tiling alongside spatial tiling to improve cache efficiency of large scale stencil computations. In a particular class of stencil computations that is the focus of their work, each time step only depends on a limited number of previous time steps and hence temporal tiling can also be incorporated.

Data reordering has been done to change the order in which the input data are processed and increase the cache efficiency. This method performs by mapping the original data indices to the new indices and has been used to improve the efficiency of both regular applications such as dense matrix kernels [15] and irregular applications such as molecular dynamic simulation and hydrodynamic computations [16]. Other methods have applied data reordering to matrix multiplication [17] and embedded multimedia processing [18] with an extra assumption that tiling is already in place.

Reducing communication overhead has also been used to speed up applications. One way to reduce communication is through asynchrony. Asynchrony can be very effective in runtime reduction as synchronizing parallel processes results in much overhead. However, asynchrony can only be used for applications whose correctness is not hurt by lack of synchronization; an example being Stochastic Gradient Descent for sparse matrix factorization [19, 20]. Computation communication overlapping has also been used to hide the communication latency in a group of problems including three-dimensional Fast Fourier Transform [21], data-parallel training of Convolutional Neural Networks [22] and distributed Stochastic Gradient Descent [23]. Block reordering is another method used in [24] to reduce the amount of communication required in Strassen's algorithm. Reducing the communication traffic has also been used to speed up applications in large scale distributed systems such as mobile networks [25].

Load balancing has been used to improve the response time of distributed applications by evenly distributing the workload among the worker units and preventing one worker to be overloaded while the other workers are idle. Among the main areas of application are cloud computing[26] and edge computing[27]. In [28] the authors have used load balancing to speed up matrix-vector multiplication in large scale machine learning and data mining applications. The authors of [29] have developed a simulator to select the best balancing technique for scientific computing applications whose stochastic and irregular natures greatly contribute to load imbalance.

Automatic data placement can also be used to improve memory efficiency of memory bound applications. These methods work by analysing the memory access patterns of a program either in compile time [30] or during the runtime using a lightweight profiling method [31] and automatically placing data segments on optimal memory components.

All of the above-mentioned methods could be used in the 1-level algorithm that is our starting point; i.e, the algorithm for which the

runtime grows faster than the workload. What we do is essentially taking the highly optimized code as a black box and adding a layer of fine-tuning on top of that.

3 Proposed Approach

3.1 General framework

In this section, we provide a general framework for speeding up any parallel code that is decomposable and whose runtime increases faster than its workload. In the subsequent sections, we will demonstrate how to apply our method to speed up the highly optimized codes for general, triangular and symmetric matrix multiplication.

We have already described our proposed method in the introduction section. Here we give a brief summary and present the pseudo codes. As we have already mentioned, our method works in multiple levels. A 2-level implementation divides the problem into subproblems and solves them using a 1-level code either serially, in parallel or using a hybrid of the both. Then the solutions to the subproblems are combined to form the general solution. Similarly, an n-level code divides the problem into subproblems that are solved using an $(n-1)$-level algorithm and then combines the results.

Our n-level algorithm for $n > 2$ is described in Algorithm 1. Given a problem A of size s, we decompose it into s/s_b subproblems of size s_b each. In lines 2-4, the subproblems are solved serially using the $(n-1)$-level algorithm (GEN_{n-1}). Then in line 5 the solutions to the subproblems are combined to form the final solution C. Note that multithreading can be incorporated in all the functions in this section and in subsections 3.2, 3.3 and 3.4. All the threads can be used to copy the subproblems (as we will explain in the next paragraph) and to combine the results. If subproblems are solved serially, then all the threads will be used to solve each subproblem. However, if b subproblems are solved in parallel, then $1/b$ of the total threads will be assigned to each subproblem. Also note that combination of each partial solution with the final result can happen at the same time that the partial solution is produced. In that case, the same number of threads used to solve each subproblem are used to combine its result with the final solution.

GEN_{n-1} in its turn calls GEN_{n-2} and this process continues until GEN_2 is called. The last level of problem decomposition happens at GEN_2 and then the subproblems are solved using GEN_1 which is the highly optimized parallel code to be sped up. In order to use the fast levels of memory hierarchy, we copy the subproblems into a fast memory and that is done in level 2 which is the level at which the actual problem solving is done. Note that in many cases the fast memory is one of the levels of cache hierarchy which we can not directly access and copy blocks into it. In those cases we still make copies of the blocks in the main memory and that causes the blocks to be automatically brought into the cache. Due to the space constraints of the fast memory, the parallel solving of subproblems happens only at level 2. And that is also why we have two sets of pseudo codes one for level $n, n > 2$ and one for level 2. Algorithm 2 describes our two-level algorithm.

Using GEN_2, a problem $\overline{A}$ of size $\overline{s}$ is divided into $\overline{s}/\overline{s_b}$ subproblems of size $\overline{s_b}$ each. $\overline{C}$ is the output and b is the degree of concurrency. Meaning that, at each iteration, the input to b of the subproblems are copied into the fast memory (line 3), the b subproblems are solved concurrently using GEN_1 (lines 4-6) and then the results are combined to form part of the final solution (line 7).

3.2 Matrix multiplication

This section illustrates our method by demonstrating how it works to speed up matrix multiplication. Software packages such as AMD Core Math Library (ACML), OpenBLAS, ATLAS and Intel Math Kernel Library (MKL) provide efficient parallel implementations of matrix multiplication as a part of the Basic Linear Algebra Subroutines (BLAS). In level 1 of our implementation, we use cblas_dgemm that is the double precision matrix multiplication code from MKL.

Algorithm 1 n-level Algorithm

1: **function** $GEN_n(A)$ ▷ Problem A of size s is decomposed into s/s_b subproblems of size s_b each. C is the output. $C_p[I]$ is the solution to the subproblem $A[I]$.
2: **for** $I = 1$ to s/s_b **do**
3: $C_p[I] \leftarrow GEN_{n-1}(A[I])$
4: **end for**
5: Combine $C_p[I], I = 1, 2, ..., s/s_b$ with C
6: **return** C
7: **end function**

Algorithm 2 Two_level Algorithm

1: **function** $GEN_2(\overline{A})$ ▷ Problem $\overline{A}$ of size $\overline{s}$ is decomposed into $\overline{s}/\overline{s_b}$ subproblems of size $\overline{s_b}$. $\overline{C}$ is the output and $\overline{C_p}[I]$ is the solution to the subproblem $\overline{A}[I]$. b is the degree of concurrency.
2: **while** $I \leq \overline{s}/(\overline{s_b} \times b)$ **do**
3: Copy b subproblems $\overline{A}[I]$ to $\overline{A}[I + b - 1]$
4: **start** b **parallel function calls**
5: $\overline{C_p}[J] \leftarrow GEN_1(\overline{A}[J])$ for $J = I, ..., (I + b - 1)$
6: **wait for all functions to return**
7: Combine $C_p[J], J = I, ..., (I + b - 1)$ with C
8: $I \leftarrow I + b$
9: **end while**
10: return $\overline{C}$
11: **end function**

Algorithm 3 *L_level* Matrix Multiplication

1: **function** $MXM_L(A, B)$ ▷ Input matrices $A_{m\times p}$, $B_{p\times n}$ and output matrix $C_{m\times n}$ are divided into blocks of sizes $s_m \times s_p$, $s_p \times s_n$ and $s_m \times s_n$ respectively.
2: **for** $I = 1$ to m/s_m **do**
3: **for** $J = 1$ to n/s_n **do**
4: **for** $K = 1$ to p/s_p **do**
5: $C[I, J] \leftarrow C[I, J] + MXM_{L-1}(A[I, K], B[K, J])$
6: **end for**
7: **end for**
8: **end for**
9: **return** C
10: **end function**

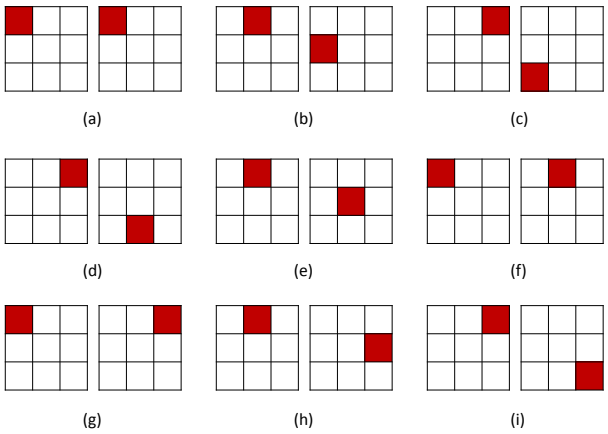

Figure 1: In parts (a)-(i) the chosen blocks in $M1$ (the matrix on the left) and $M2$ (the matrix on the right) are used to compute row 1 of $M3$. The resulting matrix $M3$ is not shown. $M3[1,1]$, $M3[1,2]$ and $M3[1,3]$ are computed in parts (a)-(c), (d)-(f) and (g)-(i) respectively. If we follow the path of the selected blocks in each matrix, a pattern similar to snake movement will result.

Our method for the matrix multiplication problem follows the general framework presented in the previous section. Here, the input is two matrices A and B of sizes $m \times p$ and $p \times n$ to be multiplied and the result is stored in matrix C of size $m \times n$. At level $L > 2$, the matrices are divided into blocks of sizes $s_m \times s_p$, $s_p \times s_n$ and $s_m \times s_n$ respectively. Product of matrix blocks $A[I,K]$ and $B[K,J]$ is computed using a lower level function and the result gets accumulated in the corresponding output block $C[I,J]$. Algorithm 3 describes our *L_level* algorithm (MXM_L).

MXM_2 is our two-level matrix-matrix multiplication algorithm. Input matrices $M1$ and $M2$ are multiplied and the result is stored in matrix $M3$. The matrix sizes are $\overline{m} \times \overline{p}$, $\overline{p} \times \overline{n}$ and $\overline{m} \times \overline{n}$ respectively.

As we previously mentioned, in level 2, the subproblems (matrix blocks in this case) are copied into the fast memory, so that the highly optimized level 1 code can benefit from the high bandwidth. To optimize the copying process, we use a serpentine pattern to iterate through the input sub-matrices. Figure 1 shows an example of this serpentine movement and how it reduces the amount of data transfer between the slow and the fast memories. Here, a 3×3 grid is used to divide matrices into smaller blocks. Degree of concurrency $b = 1$ that means one block from each matrix is selected at a time. Here, for computing $M3[1,1]$, blocks $[1,1]$, $[1,2]$ and $[1,3]$ are selected from $M1$ with their corresponding blocks from $M2$. Now, for computing $M3[1,2]$, instead of selecting blocks of $M1$ in the previous order, i.e. starting from $M1[1,1]$ and moving left-to-right, we start from $M1[1,3]$ and move right-to-left. As a result, we won't have to copy $M1[1,3]$ again as it is already in the fast memory. For $M3[1,3]$ we start from $M1[1,1]$ and move left-to-right again as then, $M1[1,1]$ is already present in the fast memory. If we consider the order in which we iterate the blocks of $M1$ and their corresponding blocks in $M2$, each of them forms a pattern like a snake movement. Using this iteration pattern we reduce the number of copied blocks by 2 for computing the first row of $M3$. The effect gets more significant for larger matrices with higher degrees of concurrency.

We should note that in our implementation, we also used serpentine pattern for MXM_L; however, as the blocks for $L-level$ matrix multiplication ($L > 2$) are usually too large to fit in the fast levels of memory hierarchy, there is not much benefit in using this pattern. Therefore, we eliminated the details from algorithm 3 to make it more concise and readable.

Our 2-level algorithm MXM_2 is described in algorithm 4. It follows the general structure of algorithm 2. Matrices $M1$, $M2$ and $M3$ of sizes $\overline{m} \times \overline{p}$, $\overline{p} \times \overline{n}$ and $\overline{m} \times \overline{n}$ are divided into blocks of sizes $\overline{s_b}_1 \times \overline{s_b}_3$, $\overline{s_b}_3 \times \overline{s_b}_2$ and $\overline{s_b}_1 \times \overline{s_b}_2$ respectively. In lines 8-15 b blocks from each of the two input matrices are copied, then in lines 16-19 b instances of our 1-level algorithm (cblas_dgemm) are called in parallel on each of the block pairs. Finally, the partial results are combined in lines 20, 21. The rest of the algorithm deals with the order in which the blocks are selected and whether or not they are already present in the fast memory (if a block is already present, then it should not be copied again). That is the serpentine pattern we previously talked about and is implemented by K_{dir} and J_{dir} that define the moving directions along the columns of $M1$ and rows of $M2$ respectively. Variables $copy_{M1}$ and $copy_{M2}$ define when the blocks of $M1$ and $M2$ should be copied.

Algorithm 4 Two_level Matrix Multiplication

1: **function** $MXM_2(M1, M2)$ ▷ Input matrices $M1_{\overline{m}\times\overline{p}}$ and $M2_{\overline{p}\times\overline{n}}$ and the output matrix $M3_{\overline{m}\times\overline{n}}$ are divided into blocks of sizes $\overline{s_b}_1 \times \overline{s_b}_3$, $\overline{s_b}_3 \times \overline{s_b}_2$ and $\overline{s_b}_1 \times \overline{s_b}_2$ respectively. b is the degree of concurrency. $C_p[U]$ stores the partial result of $M1[I,U] \times M2[U,J]$. MXM_1 is cblas_dgemm() in our case.
2: $copy_{M1} \leftarrow true$, $copy_{M2} \leftarrow true$
3: $K \leftarrow 1$, $J \leftarrow 1$, $K_{dir} \leftarrow 1$, $J_{dir} \leftarrow 1$
4: **for** $I = 1$ to $\overline{m}/\overline{s_b}_1$ **do**
5: **for** $\overline{n}/\overline{s_b}_2$ iterations involving J **do**
6: **for** $\overline{p}/\overline{s_b}_3$ iterations involving K **do**
7: **if** $copy_{M1}$ **then**
8: Copy b blocks from $M1$:
9: $(M1[I,K]$ to $M1[I, K + (b-1) \times K_{dir}])$
10: **end if**
11: **if** $copy_{M2}$ **then**
12: copy b blocks from $M2$:
13: $(M2[K,J]$ to $M[K + (b-1) \times K_{dir}, J])$
14: **end if**
15: **start** b **parallel function calls**
16: $C_p[U] \leftarrow MXM_1(M1[I,U], M2[U,J])$
17: for $U = K, ..., K + (b-1) \times K_{dir}$
18: **wait for all functions to return**
19: $M3[I,J] \leftarrow M3[I,J] + C_p[K] +$
20: $... + C_p[K + (b-1) \times K_{dir}]$
21: $K \leftarrow K + b \times K_{dir}$; $copy_{M1} \leftarrow true$
22: **end for**
23: $K_{dir} \leftarrow -K_{dir}$; $copy_{M1} \leftarrow false$
24: $J \leftarrow J + J_{dir}$; $copy_{M2} \leftarrow true$
25: **end for**
26: $J_{dir} \leftarrow -J_{dir}$; $copy_{M2} \leftarrow false$; $copy_{M1} \leftarrow true$
27: **end for**
28: **return** $M3$
29: **end function**

After all blocks corresponding to the K variable in the block

matrix multiplication formula

$$M3[I,J] = \sum_{K=1}^{\overline{p}/\overline{s_{b3}}} M1[I,K] \times M2[K,J] \quad (1)$$

have been used, we change the direction in which we select those blocks for computing $M3[I, J+1]$. That is when blocks from $M1$ are reused. Similarly, after $M3[I,1]...M3[I,\overline{n}/\overline{s_{b2}}]$ have all been computed; we compute $M3[I+1,J]$s in reverse direction and b blocks from $M2$ are reused.

3.3 Triangular Matrix Multiplication

This section illustrates how our methodology may be employed to multiply two matrices where one of the matrices is (upper/lower) triangular. The highly optimized function we use at the first level is cblas_dtrmm from Intel MKL.

Algorithm 5 *L_level* Triangular Matrix Multiplication

function $TrMXM_L(A,B)$ ▷ Input matrices $A_{m\times m}$, $B_{m\times n}$ and output matrix $C_{m\times n}$ are divided into blocks of sizes $s_{b1}\times s_{b1}$, $s_{b1}\times s_{b2}$ and $s_{b1}\times s_{b2}$ respectively. b is the number of block multiplications done in parallel.
  for $I = 1$ to m/s_{b1} **do**
    for $J = 1$ to n/s_{b2} **do**
      for $K = 1$ to I **do**
        if $I == K$ **then**
          $C[I,J] \leftarrow C[I,J] +$
          $TrMXM_{L-1}(A[I,K], B[K,J])$
        else
          $C[I,J] \leftarrow C[I,J] + MXM(A[I,K], B[K,J])$
        end if
      end for
    end for
  end for
  return C
end function

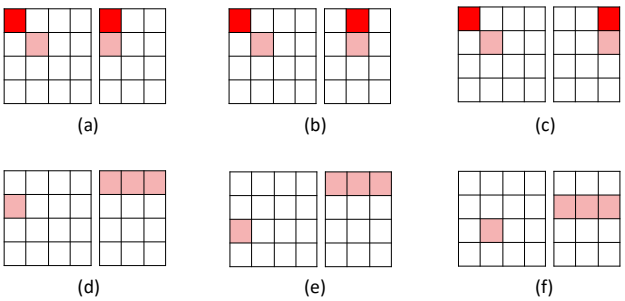

Figure 2: Phase 1 and 2 of $TrMXM_2$. Parts (a)-(c) show three steps from phase 1 where at each step two triangular blocks from the matrix $M1$ are multiplied in parallel to the corresponding blocks of $M2$. Each pair of blocks that are multiplied at each step are shown with the same shade. The resulting matrix $M3$ is not shown. Parts (d)-(f) show three steps from phase 2 where an ordinary block from $M1$ is multiplied concurrently to three blocks from $M2$. For all the parts (a)-(f), $M1$ is the matrix on the left, $M2$ is the matrix on the right and $M3$ is not shown.

Algorithm 6 Two_level Triangular Matrix Multiplication

function $TrMXM_2(M1,M2)$ ▷ Input matrices $M1_{\overline{m}\times\overline{m}}$ and $M2_{\overline{m}\times\overline{n}}$ and the output matrix $M3_{\overline{m}\times\overline{n}}$ are divided into blocks of sizes $\overline{s_{b1}}\times\overline{s_{b1}}$, $\overline{s_{b1}}\times\overline{s_{b2}}$ and $\overline{s_{b1}}\times\overline{s_{b2}}$ respectively. $b1$ is the number of triangular block multiplications done in parallel and $b2$ is the number of ordinary block multiplications done in parallel. $TrMXM_1$ is cblas_dtrmm() in our case.
  for $I = 1$ to $\overline{m}/\overline{s_{b1}}$ increment = $b1$ **do** ▷ phase 1
    for $J = 1$ to $\overline{n}/\overline{s_{b2}}$ **do**
      Copy $b1$ blocks of $M1$ along the main diagonal:
        i.e. $M1[I,I]$ to $M1[I+b1-1, I+b1-1]$
      Copy $b1$ blocks of $M2$ along column J:
        i.e. $M2[I,J]$ to $M2[I+b1-1, J]$.
      start $b1$ **parallel function calls**
        $C[U,J] \leftarrow C[U,J] +$
          $TrMXM_1(M1[U,U], M2[U,J])$
        for $U = I, ..., (I+b1-1)$
      wait for all functions to return
    end for
  end for
  $cnt \longleftarrow 0$
  for $I = 2$ to $\overline{m}/\overline{s_{b1}}$ **do** ▷ phase 2
    for $J = 1$ to $I-1$ **do**
      for $K = 1$ to $\overline{n}/\overline{s_{b2}}$ **do**
        Copy $M1[I,J]$; Copy $M2[J,K]$; $cnt \longleftarrow cnt+1$
        if $cnt == b2$ **then**
          start $b2$ **parallel function calls**
            Call $b2$ instances of $\overline{MXM}$ over
            blocks copied from $M1$ and $M2$
          wait for all functions to return
          Sum up partial results of $\overline{MXM}$
            to $b2$ corresponding blocks of $M3$
          $cnt \longleftarrow 0$
        end if
      end for
    end for
  end for
  return $M3$
end function

Without loss of generality, we assume that the triangular matrix appears at the left side of multiplication. Therefore, in our triangular matrix multiplication problem, matrices A and B of sizes $m\times m$ and $m\times n$ respectively, are multiplied resulting in a matrix C of size $m\times n$. Notice that A being triangular means that it should be a square matrix. Again, without loss of generality, we assume A is lower triangular.The general framework is also applicable here. However, the nature of the problem implies some modifications in our implementation. Here, we divide A into blocks of size $s_{b1}\times s_{b1}$. If $g = \frac{m}{s_{b1}}$, then we will have $g\times g$ blocks, g of which are triangular, $\frac{g\times g-g}{2}$ of which are ordinary (non _ triangular) blocks and $\frac{g\times g-g}{2}$ of which are zero blocks.The zero blocks are ignored while the ordinary blocks are involved in ordinary matrix multiplication and the triangular blocks get involved in triangular matrix multiplication.

Algorithm 5 describes the *L_level* algorithm ($TrMXM_L$) where $L > 2$. Here, we break A, B and C into blocks of sizes $s_{b1}\times s_{b1}$,

$s_{b1} \times s_{b2}$ and $s_{b1} \times s_{b2}$ respectively. In lines 6, 7 the triangular blocks from A are multiplied to the corresponding blocks from B whereas in line 9 the ordinary blocks from A are multiplied to the corresponding blocks from B. Here, MXM is an efficient presumably multilevel matrix multiplication algorithm.

For our two-level algorithm we make an observation: cblas_dtrmm that is used at the leaf level of triangular matrix multiplication is optimized to benefit from the lower amount of computations that result from the zero elements. Therefore, we can further improve the runtime by grouping the triangular blocks together and the ordinary blocks together. That way, the more efficient triangular matrix multiplications should not wait for the results of ordinary matrix multiplications. At each iteration, we operate either on $b1$ triangular block multiplications or on $b2$ ordinary block multiplications in parallel where $b1$ can be different from $b2$.

Our algorithm uses two phases in order to correctly select and group together the matrix blocks. In the first phase, $b1$ triangular blocks from the main diagonal of $M1$ are copied and at each step, they are multiplied concurrently to one other block from $M2$. In the second phase, ordinary blocks from $M1$ are picked in a top-to-bottom and left-to-right order. For each block x from $M1$, we iterate over all blocks of $M2$ to which x should be multiplied. When the total number of block pairs reaches $b2$, we stop and perform $b2$ multiplications concurrently.

The above-mentioned process is illustrated in figure 2. The top row shows three steps from phase 1 and the bottom row shows three steps from phase 2. In each part, $M1$ is the matrix on the left and $M2$ is the matrix on the right where a 4×4 grid is used to block $M1$ and a 4×3 grid is used to block $M2$. Here, $b1 = 2$ and $b2 = 3$. In part (a), two triangular blocks $M1[1, 1]$ and $M2[2, 2]$ are copied to the fast memory; and through steps (a) - (c), those blocks are concurrently multiplied to $M2[1, J]$ and $M2[2, J]$ respectively ($J = 1, 2, 3$). In part (d), the ordinary block $M1[2, 1]$ is copied three times in the fast memory and three parallel multiplications are done on $M1[2, 1]$ and $M2[1, J], J = 1, 2, 3$. In parts (e) and (f) the next ordinary blocks from $M1$ are chosen in the top-to-bottom and left-to-right order and multiplied concurrently with three blocks of $M2$. Note that if the degree of concurrency for the ordinary blocks ($b2$) was equal to 4, then in part (d), block $M1[3, 1]$ would also be copied along with a second copy of block $M2[1, 1]$. Then there would be 4 parallel multiplications, three of them involving $M1[2, 1]\times M2[1, J](J = 1, 2, 3)$ and one of them $M1[3, 1] \times M2[1, 1]$. The cases for parts (e) and (f) would be similar.

Algorithm 6 describes our 2-level triangular matrix multiplication algorithm, $TrMXM_2$. We divide the input matrices $M1_{\overline{m}\times\overline{m}}$ and $M2_{\overline{m}\times\overline{n}}$ into blocks of sizes $\overline{s_{b1}} \times \overline{s_{b1}}$ and $\overline{s_{b1}} \times \overline{s_{b2}}$ respectively. The output matrix $M3_{\overline{m}\times\overline{n}}$ will also be divided into blocks of size $\overline{s_{b1}} \times \overline{s_{b2}}$. Lines 2-14 implement the first phase of the algorithm where we run $b1$ parallel cblas_dtrmm instances on the triangle-rectangle block multiplications. Lines 15-33 implement the second phase of our algorithm where we run $b2$ parallel instances of $\overline{MXM}$ over the square-rectangle block multiplications where $\overline{MXM}$ is an efficient matrix multiplication algorithm. The auxiliary variable *cnt* is used to keep track of the square blocks being multiplied in parallel.

3.4 Symmetric Matrix Multiplication

In this section, we illustrate our methodology on symmetric matrix multiplication. This is when one of the matrices involved in the multiplication is symmetric. The highly optimized parallel function we use at the first level is cblas_dsymm from Intel MKL.

Without loss of generality, we assume that the symmetric matrix appears at the left side of multiplication. Therefore, the problem can be stated as the following: matrices A and B of sizes $m \times m$ and $m \times n$ respectively where A is a symmetric matrix, are multiplied and the result is stored in matrix C of size $m \times n$. Notice again that symmetry is only defined for square matrices; that is why A is $m \times m$. The same methodology is used as the general framework. Here again, similar to triangular multiplication, the nature of the problem implies some modifications to the implementation details. We divide A into blocks of size $s_{b1} \times s_{b1}$. If $g = \frac{m}{s_{b1}}$, then we will have $g \times g$ blocks, g of which are symmetric. The ordinary blocks are involved in ordinary matrix multiplication and the symmetric blocks get involved in symmetric matrix multiplication.

Algorithm 7 describes the *L_level* algorithm ($SymMXM_L$) where $L > 2$. Here, we break A, B and C into blocks of sizes $s_{b1} \times s_{b1}$, $s_{b1} \times s_{b2}$ and $s_{b1} \times s_{b2}$ respectively. In lines 6, 7 the symmetric blocks from A are multiplied to the corresponding blocks from B whereas in line 9 ordinary blocks from A are multiplied to the corresponding blocks from B. Here, similar to that of triangular matrix multiplication, MXM is an efficient matrix multiplication algorithm that could be multilevel.

Algorithm 7 *L_Level* Symmetric Matrix Multiplication

1: **function** $SymMXM_L(A, B)$ ▷ Input matrices $A_{m\times m}$, $B_{m\times n}$ and output matrix $C_{m\times n}$ are divided into blocks of sizes $s_{b1} \times s_{b1}$, $s_{b1} \times s_{b2}$ and $s_{b1} \times s_{b2}$ respectively.
2: **for** $I = 1$ to m/s_{b1} **do**
3: **for** $J = 1$ to n/s_{b2} **do**
4: **for** $K = 1$ to m/s_{b1} **do**
5: **if** $I == K$ **then**
6: $C[I, J] \leftarrow C[I, J] +$
7: $SymMXM_{L-1}(A[I, K], B[K, J])$
8: **else**
9: $C[I, J] \leftarrow C[I, J] + MXM(A[I, K], B[K, J])$
10: **end if**
11: **end for**
12: **end for**
13: **end for**
14: **return** C
15: **end function**

Algorithm 8 describes our 2-level symmetric matrix-matrix multiplication algorithm, $SymMXM_2$. We divide $M1_{\overline{m}\times\overline{m}}$ and $M2_{\overline{m}\times\overline{n}}$ into blocks of sizes $\overline{s_{b1}} \times \overline{s_{b1}}$ and $\overline{bs_1} \times \overline{bs_2}$ respectively. The resulting matrix $M3_{\overline{m}\times\overline{n}}$ will also be divided into blocks of size $\overline{s_{b1}} \times \overline{s_{b2}}$. Similar to triangular matrix multiplication, we group the symmetric blocks together and the ordinary blocks together. The algorithm works in two phases where at each step of the first phase, $b1$ symmetric blocks multiplications and at each step of the second phase $b2$ ordinary block multiplications are performed in parallel. $b1$ and $b2$ are not necessarily equal. Lines 2-14 implement the first phase

and lines 16-33 implement the second phase of our algorithm. The auxiliary variable *cnt* is used to keep track of the ordinary blocks being multiplied in parallel. $\overline{MXM}$ is an efficient matrix multiplication algorithm.

Algorithm 8 Two_level Symmetric Matrix Multiplication

1: **function** $SymMXM_2(M1, M2)$ ▷ Input matrices $M1_{\overline{m}\times\overline{m}}$ and $M2_{\overline{m}\times\overline{n}}$ and the output matrix $M3_{\overline{m}\times\overline{n}}$ are divided into blocks of sizes $\overline{s_{b1}} \times \overline{s_{b1}}$, $\overline{s_{b1}} \times \overline{s_{b2}}$ and $\overline{s_{b1}} \times \overline{s_{b2}}$ respectively. $b1$ is the number of symmetric block multiplications done in parallel and $b2$ is the number of ordinary block multiplications done in parallel. $SymMXM_1$ is cblas_dsymm() in our case.
2: **for** $I = 1$ to $\overline{m}/\overline{s_{b1}}$ increment $= b1$ **do** ▷ phase 1
3: **for** $J = 1$ to $\overline{n}/\overline{s_{b2}}$ **do**
4: Copy $b1$ blocks of $M1$ along the main diagonal:
5: i.e. $M1[I, I]$ to $M1[I + b1 - 1, I + b1 - 1]$
6: Copy $b1$ blocks $M2$ along column J:
7: i.e. $M2[I, J]$ to $M2[I + b1 - 1, J]$.
8: **start** $b1$ **parallel function calls**
9: $C[U, J] \leftarrow C[U, J] +$
10: $SymMXM_1(M1[U, U], M2[U, J])$
11: for $U = I, ..., (I + b1 - 1)$
12: **wait for all functions to return**
13: **end for**
14: **end for**
15: $cnt \longleftarrow 0$
16: **for** $I = 1$ to $\overline{m}/\overline{s_{b1}}$ **do** ▷ phase 2
17: **for** $J = 1$ to $\overline{m}/\overline{s_{b1}}, I \neq J$ **do**
18: **for** $K = 1$ to $\overline{n}/\overline{s_{b2}}$ **do**
19: Copy $M1[I, J]$
20: Copy $M2[J, K]$
21: $cnt \longleftarrow cnt + 1$
22: **if** $cnt == b2$ **then**
23: **start** $b2$ **parallel function calls**
24: Call $b2$ instances of $\overline{MXM}$ over
25: blocks copied from $M1$ and $M2$
26: **wait for all functions to return**
27: Sum up partial results of $\overline{MXM}$
28: to $b2$ corresponding blocks of $M3$
29: $cnt \longleftarrow 0$
30: **end if**
31: **end for**
32: **end for**
33: **end for**
34: **return** $M3$
35: **end function**

3.5 Performance Analysis

In this section, a simple shared-memory model in which processes communicate by reading/writing from/to a shared memory is used to describe the runtime of the parallel codes. Using this model, we formulate the conditions under which our method will be effective and also the speed up we can expect. Let $T^i_{x,y}$ be the time our algorithm takes at level i to solve a problem of size y using x processors. Table 2 illustrates our notation for a level 1 code. Here, b and c are integer multipliers.

Table 2: Notations used for the runtime of 1-level algorithm depend on the problem size and the number of processors.

	p processors	bp processors
problem size= n	$T^1_{p,n}$	$T^1_{bp,n}$
problem size= cn	$T^1_{p,cn}$	$T^1_{bp,cn}$

Let,

$$r = \frac{T^1_{bp,cn}}{T^1_{p,cn}} \tag{2}$$

$$k = \frac{T^1_{p,cn}}{f(c) \times T^1_{p,n}} \tag{3}$$

Where $f(c)$ is defined as follows: if we decompose a problem of size cn into subproblems of size n; then $f(c)$ denotes the number of those subproblems. In the case of matrix multiplication, for example, if the initial problem is multiplying two $2n \times 2n$ matrices and we break each matrix into four matrices of size $n \times n$; then we should multiply eight matrix pairs of size $n \times n$ each. Here, we have $c = 2$ and $f(c) = 8$.

Let $T_i = T^i_{bp,cn}$. We wish to find the ratio T_i/T_1. We consider two cases: $i = 2$ and $i > 2$. The reason we consider two cases is that we use parallelism only at the second level.

We define $g(c, n)$ to be the time it takes to combine $f(c)$ subproblems of size n each. For a 2-level algorithm, if the subproblems run serially, then all the bp processors can be used for each of them. We will have:

$$T^2_{bp,cn} = f(c)T^1_{bp,n} + g(c, n) \tag{4}$$

However, if we run b subproblems in parallel at a time, then only p processors can be used for each of the subproblems. In that case, the runtime equation will be:

$$T^2_{bp,cn} = \frac{f(c)}{b}T^1_{p,n} + g(c, n) \tag{5}$$

In the rest of this subsection, we use equation 5 to describe T_2. That is

$$T_2 = \frac{f(c)}{b}T^1_{p,n} + g(c, n)$$

Therefore,

$$\frac{T_2}{T_1} = \frac{1 + rbk\delta}{rbk} \tag{6}$$

where

$$\delta = \frac{g(c, n)}{T_1} \tag{7}$$

For level $i > 2$, we consider c_1 and c_2 to be divisors of c, i.e. $c = c_1c_2$. The time to solve a problem of size $cn = c_1c_2n$ using bp processors and an i-level algorithm will be:

$$T_i = T^i_{bp,cn} = f(c_1)T^{i-1}_{bp,c_2n} + g(c_1, c_2n) \tag{8}$$

where $g(c_1, c_2n)$ is the time to combine $f(c_1)$ subproblems of size c_2n.

3.6 Speed Up Analysis

In this section, we formulate the conditions for when increasing the number of levels results in speed up. Throughout this section, we consider the problem size to be cn and the total number of available processors to be bp. Also, as mentioned before, we use different sets of conditions for T_2 and $T_i, i > 2$ and the reason is that we only run the subproblems in parallel at level 2.

3.6.1 $T_2 < T_1$

The condition for a 2-level algorithm to be faster than a 1-level algorithm is as follows:

$$T_2 < T_1 \Rightarrow \frac{T_2}{T_1} < 1$$

$$\Rightarrow \frac{1 + rbk\delta}{rbk} < 1 \Rightarrow rbk(1-\delta) > 1 \tag{9}$$

In the formulation for δ, we assume that the combination time is negligible compared to T_1 i.e. $g(c,n) \ll T_1$. Then $\delta \to 0$ and the condition for $T_2 < T_1$ will be:

$$rbk > 1 \tag{10}$$

The expected speed up is ${}^{T_1}/_{T_2} = rbk$ (as $\delta \to 0$).

3.6.2 $T_i < T_1 (i > 2)$

Using equation (8) we get the following condition for an i-level algorithm ($i > 2$) to be faster than a 1-level algorithm:

$$T_i < T_1 \Rightarrow \frac{T_i}{T_1} < 1$$

$$\Rightarrow f(c_1)\,\frac{T^{i-1}_{bp,c_2n}}{T_1} + \frac{g(c_1,c_2n)}{T_1} < 1 \tag{11}$$

Again, assuming $g(c_1, c_2n) \ll T_1$, the inequality (11) will reduce to

$$f(c_1)\,\frac{T^{i-1}_{bp,c_2n}}{T_1} < 1 \tag{12}$$

Using equations 5 and 8 for $i = 3$ we get:

$$T^{i-1}_{bp,c_2n} = \frac{f(c_2)}{b} T^{i-2}_{p,n} + g(c_2,n) \tag{13}$$

$$T_{i-1} = T^{i-1}_{bp,cn} = \frac{f(c_1c_2)}{b} T^{i-2}_{p,n} + g(c_1c_2,n) \tag{14}$$

and for $i > 3$

$$T^{i-1}_{bp,c_2n} = f(c_2) T^{i-2}_{bp,n} + g(c_2,n) \tag{15}$$

$$T_{i-1} = T^{i-1}_{bp,cn} = f(c_1c_2) T^{i-2}_{bp,n} + g(c_1c_2,n) \tag{16}$$

We make the assumption that $f(.)$ is separable, meaning that $f(c_1c_2) = f(c_1)f(c_2)$. Assuming that $g(c_2,n)$ and $g(c_1c_2,n)$ are negligible compared to T_1; for all values of $i > 2$ we will have:

$$f(c_1)\,\frac{T^{i-1}_{bp,c_2n}}{T_1} = \frac{T_{i-1}}{T_1} \tag{17}$$

Combining equations (17) and (12), we get

$$\frac{T_{i-1}}{T_1} < 1 \tag{18}$$

that is (by induction) $rbk > 1$.

3.6.3 $T_i < T_{i-1} (i > 3)$

In the previous section, the condition for $T_i < T_1$ was established. Even if $T_i < T_1$, it may be better to stay at level $i-1$ and not to increase the level. To decide whether or not we can use another level, we can consider the $(i-1)$-level code as a new 1-level code. Then recalculate r, k and δ for this code and set $b = 1$ (that is because we don't use parallelism for $i > 2$). We call these newly computed values r', k', δ' and b'. Then the condition for $T_i < T_{i-1}$ will be $r'b'k' > 1$. As $b' = 1$ the condition will reduce to $r'k' > 1$.

3.7 Performance Analysis: Matrix Multiplication

The general model can be applied to matrix multiplication. Without loss of generality, we assume the matrices are square $cn \times cn$ matrices divided into blocks of size $n \times n$. The model details will be as follows:

- $f(c) = c^3$ therefore by plugging it in (3) we get

$$k = \frac{T^1_{p,cn}}{c^3 T^1_{p,n}} \tag{19}$$

- $g(c,n) = (c-1)c^2a(n)$. Here $a(n)$ is the time it takes to add two matrix blocks of size $n \times n$ and we should add $(c-1)c^2$ of them to get the final result for a $cn \times cn$ matrix multiplication. By plugging it in (7) we will have:

$$\delta = \frac{(c-1)c^2a(n)}{T_1} \tag{20}$$

Therefore, for matrix multiplication, equations (5) and (8) will be:

$$T^2_{bp,cn} = \frac{c^3}{b} T^1_{p,n} + (c-1)c^2a(n) \tag{21}$$

$$T_i = {c_1}^3 T^{i-1}_{bp,c_2n} + (c_1-1)c_1^2 a(c_2n) \tag{22}$$

As $f(c)$ is separable and the addition time is negligible compared to multiplication time for large enough matrices (and therefore we can safely assume that $\delta \to 0$), all the speed up analysis remains valid.

3.8 Performance Analysis: Triangular Matrix Multiplication

Our multilevel method for triangular matrix multiplication is a compound method. Meaning that for solving the subproblems, we should call both triangular matrix multiplication and ordinary matrix multiplication functions on the smaller instances. Consider the following equations for $T^2_{bp,cn}$ and $T^i_{bp,cn} (i > 2)$ respectively, where - as before - $c = c_1c_2$:

$$T^2_{bp,cn} = \frac{f_1(c)}{b1} T^1_{p_1,n} + \frac{f_2(c)}{b2} \tau_{p_2,n} + g(c,n) \tag{23}$$

$$\begin{aligned} T^i_{bp,cn} &= f_1(c_1) T^{i-1}_{bp,c_2n} \\ &+ f_2(c_1)\tau_{bp,c_2n} + g(c_1,c_2n) \end{aligned} \tag{24}$$

In both (23) and (24) $f_1(.)$ is the number of triangular matrix multiplication subproblems and $f_2(.)$ is the number of ordinary matrix multiplication subproblems. $b1$ is the degree of concurrency for triangular block multiplication and $b2$ is the degree of concurrency for ordinary block multiplication. $p_1 = {}^{(bp)}/_{b1}$ and $p_2 = {}^{(bp)}/_{b2}$. We use only one $g(.,.)$ function because the result of multiplying a triangular block with an ordinary block will not necessarily be triangular and therefore there is no need to distinguish between the combination times. Also, $\tau_{x,y}$ is the runtime of an ordinary matrix multiplication function on blocks of size y using x processors.
Let,

$$\theta = \frac{b2}{b1} \tag{25}$$

$$\lambda = \frac{\tau_{p_2,n}}{T^1_{p_1,n} \times \theta} \tag{26}$$

Now, if we define:

$$b = b1 \tag{27}$$

$$f(x) = f_1(x) + \lambda f_2(x) \tag{28}$$

equations (23) and (24) will turn to

$$T^2_{bp,cn} = \frac{f(c)}{b} T^1_{p,n} + g(c,n) \tag{29}$$

$$T^i_{bp,cn} = f(c_1) T^{i-1}_{bp,c_2 n} + g(c_1, c_2 n) \tag{30}$$

that are the same as equations 5 and 8. Again, without loss of generality, we assume that the matrices are square, their sizes are $cn \times cn$ and their block sizes are $n \times n$. As a result,

- $f_1(c) = c^2$ and $f_2(c) = \frac{c^3 - c^2}{2}$. By using (28) we get:

$$f(c) = \frac{\lambda c^3 + (2 - \lambda)c^2}{2} \tag{31}$$

- $g(c,n) = \frac{c^3 - c^2}{2} a_n$ where a_n is the time to add two $n \times n$ blocks.

We can safely assume that $g(c,n) \ll T_1$ for large enough matrices and therefore the condition for $T_2 < T_1$ still holds. However, $f(c)$ is not separable, therefore the conditions for $T_i < T_1$ for $i > 2$ will not necessarily work.

3.9 Performance Analysis: Symmetric Matrix Multiplication

The performance analysis for symmetric matrix multiplication has a lot in common with the performance analysis for triangular matrix multiplication.Equations 23 - 30 are valid for symmetric matrix multiplication. The definitions for $f_1(.)$, $f_2(.)$, $b1$, $b2$, θ and λ are the same as those in section 3.8 and the same equations $b = b1$ and $f(x) = f_1(x) + \lambda f_2(x)$ work for symmetric matrices. The differences are in details as described in the following (again, without loss of generality, we assume that the matrices to be multiplied are both square matrices of sizes $cn \times cn$ which are divided into blocks of sizes $n \times n$):

- $f_1(c) = c^2$ and $f_2(c) = c^3 - c^2$. By using (28) we get:

$$f(c) = \lambda c^3 + (1 - \lambda)c^2 \tag{32}$$

- $g(c,n) = c^2(c-1)a_n$ with a_n being the time to add two $n \times n$ blocks.

Once again, we can assume that $g(c,n) \ll T_1$ for large enough matrices and therefore the condition for $T_2 < T_1$ holds. However, the function $f(c)$ is only separable if $\lambda = 1$ and that is the only case where the conditions $T_i < T_1 (i > 2)$ will work.

4 KNL Architecture

4.1 Architecture Overview

Knights Landing (KNL) [32] is the codename for the second generation Intel Xeon Phi product family. It is a many-core architecture enabling highly parallel workloads. KNL CPU includes up to 36 active tiles. Each tile includes 2 cores, 2 vector processing units per core and one L_2 cache shared between the cores. A 2D mesh interconnect provides connection between the tiles, memory controllers and other on-board elements. The mesh supports the MESIF (modified, exclusive, shared, invalid, forward) cache-coherence protocol and uses a distributed tag directory to keep all the L_2 caches coherent in all the tiles.

KNL includes 2 types of memory: 16GB of multichannel DRAM (MCDRAM) and up to 384 GB of double data rate (DDR) memory. MCDRAM is a high bandwidth memory (HBM) that provides an aggregate bandwidth of more than 450 GBps. The aggregate bandwidth of DDR is more than 90 GBps.

4.2 Memory Modes

The two memory modules can be configured in three modes explained below:

4.2.1 Flat Mode

In this mode, MCDRAM is treated as an addressable memory alongside DDR. Flat mode gives the user the ability to allocate data either from DDR or MCDRAM. The downside is that it requires software modification.

4.2.2 Cache Mode

In cache mode, MCDRAM is configured as a memory-side cache for the whole DDR. Here, the user has no control over MCDRAM usage, but no software modification is required.

4.2.3 Hybrid Mode

This mode is a combination of flat mode and cache mode. In hybrid mode, a portion of MCDRAM (either 0.25 or 0.5) is used in cache mode and the remaining portion is used in flat mode.

4.3 KNL Clustering Modes

The mesh interconnect provides five different clustering modes. Each of these clustering modes defines the affinity properties of tiles, tag directories and memory controllers. These modes are explained in the following:

4.3.1 All-to-All

This is the most general mode that lacks any affinity between tiles, directory and memory.

4.3.2 Quadrant

This mode divides the KNL chip into four virtual quadrants. A request from any tile can go to any directory, but the directory can only access the memory in its own quadrant. Meaning that this mode provides affinity between directory and memory. However, there is no affinity between a tile and the directory or memory.

4.3.3 Hemisphere

This mode is similar to the quadrant mode, with the only difference that the chip is divided into 2 hemispheres instead of 4 quadrants.

4.3.4 SNC4

SNC stands for Sub-NUMA Clustering. SNC4 is a more restricted version of the quadrant mode where the quadrants are viewed by the OS as nonuniform memory access (NUMA) nodes (or clusters). Here, a request form a tile accesses the directory in the same cluster and the directory will also access a memory controller in that same cluster. Meaning that this mode provides affinity between tiles, memory directories and memory controllers.

4.3.5 SNC2

This mode is similar to SNC4 the only difference being the number of clusters. Here the number of clusters is two instead of four.

5 Experimental Results

We implemented our algorithms using C++ and OpenMP. The total number of threads was set to 64. Using a larger number of threads led to performance degradation. We used the Intel icpc compiler with the flags setting "-O3 -xMIC-AVX512 -mkl -lmemkind -qopenmp". In "flat" memory mode, we have to manually allocate memory from MCDRAM. This was done using "hbw_posix_memalign" command. To allocate memory from DDR, the command "posix_memalign" was used.

For all three of our algorithms, namely matrix-matrix multiplication, triangular matrix-matrix multiplication and symmetric matrix-matrix multiplication, we used double precision square matrices as test data for both the operands. For matrix pairs of sizes up to $16K \times 16K$, the reported times are the average over 10 run times. For matrix pairs of sizes $32K \times 32K$ and $64K \times 64K$, we used only 5 runs. That is because those runs were very time consuming and also because at those sizes, the runtime values were quite stable and 5 runs seemed to be sufficient. The error bound for our reported times is at most 4%.

With 3 memory modes and 5 clustering modes, there are 15 possible architectural configurations in the KNL. Of these, the *cache, all-to-all* mode is not supported. So, in reality, the KNL has 14 architectural configurations. In our experiments, we do not consider the hybrid memory mode. This leaves us with 9 configurations to consider. There are at least three different scenarios for configuration selection; the selection of scenario being limited by system and application constraints.

1. S1: Each algorithm can select the configuration to run on based on the size of the matrices to be multiplied.
2. S2: The configuration is determined by the application and cannot be changed by the algorithm.
3. S3: Each algorithm must use the same configuration for all instances; different algorithms can use different configurations.

In Sections 5.1-5.3, we give the measured average run times for the three versions of matrix multiplication considered in this paper for all 9 of the architectural configurations considered. A comparison of these run times for each of the three scenarios (S1-S3) of configuration selection is done in Section 5.4. The speed up values reported in Section 5.4 are the percentage of runtime reduction $(T_{old}-T_{new})/T_{old}$.

5.1 Matrix Multiplication Times

Table 3 shows the runtime values for the code we use as the 1-level matrix multiplication algorithm (cblas_dgemm) for different matrix sizes and different KNL memory/clustering configurations.

Table 4 contains the runtimes for our 2-level algorithm. For all the matrix sizes, the input and output matrices are divided into sub-matrices using a 4×4 grid. The degree of concurrency used in our 2-level implementation is ($b = 4$), meaning that at each step 4 pairs of matrix blocks are multiplied in parallel. We have also experimented with other block arrangements such as 2×2 and other degrees of concurrency such as 2 and 8; however, the current configuration resulted in the best runtimes.

Tables 5 and 6 show the runtimes for our 3 and 4-level algorithms respectively. For both the levels, we used 2×2 grids to divide the input and output matrices into sub-matrices. The block pairs were multiplied sequentially.

5.2 Triangular Matrix Multiplication Times

Table 7 shows the run times for the one-level code (cblas_dtrmm). This table together with Table 3 show the speed up potential using our method. Consider $64K \times 64K$ multiplications as an example. Instead of performing one $64K \times 64K$ triangular matrix multiplication; we can multiply six $32K \times 32K$ blocks and combine the results. Out of those block multiplications, four of them are triangular matrix multiplication and two of them are ordinary matrix multiplication. If we consider the combination time to be small enough, then on all the memory-clustering configurations, the time to sequentially perform two ordinary multiplications of $32K \times 32K$ matrices plus the time to sequentially perform four triangular multiplications of $32K \times 32K$ matrices is less than the time to perform one $64K \times 64K$ triangular matrix multiplication. That notifies us about the potential of speed up. Note that this is just an initial evaluation and in practice we can also use parallelism and different settings for block sizes.

Table 8 shows the run time values for our two-level algorithm. Here, matrix A is divided using a 2×2 grid and matrices B and C are divided using a 2×1 block arrangement. Degree of concurrency for

Table 3: Run time (in seconds) of the 1-level matrix multiplication using cblas_dgemm

Matrix Dimensions	Flat-SNC4	Flat-SNC2	Flat-All2All	Flat-Quadrant	Flat-Hemisphere
$4K \times 4K$	0.65	0.39	0.30	0.33	0.30
$8K \times 8K$	3.41	1.99	1.64	1.49	**1.39**
$16K \times 16K$	23.21	15.42	12.74	11.92	**10.25**
$32K \times 32K$	119.81	121.47	102.53	101.58	102.63
$64K \times 64K$	1002.30	843.25	843.04	826.23	824.15
Matrix Dimensions	Cache-SNC4	Cache-SNC2	Cache-All2All (not supported)	Cache-Quadrant	Cache-Hemisphere
$4K \times 4K$	0.48	0.42	—	0.32	**0.29**
$8K \times 8K$	1.94	1.71	—	1.51	**1.39**
$16K \times 16K$	10.89	11.32	—	12.32	10.36
$32K \times 32K$	88.90	87.39	—	97.52	**80.18**
$64K \times 64K$	**618.26**	781.83	—	871.10	652.60

Table 4: Run time (in seconds) of the 2-level matrix multiplication algorithm

Matrix Dimensions	Flat-SNC4	Flat-SNC2	Flat-All2All	Flat-Quadrant	Flat-Hemisphere
$4K \times 4K$	0.36	0.32	0.30	0.30	**0.28**
$8K \times 8K$	2.42	1.52	1.27	1.21	1.25
$16K \times 16K$	113.51	14.82	10.12	8.87	8.85
Matrix Dimensions	Cache-SNC4	Cache-SNC2	Cache-All2All (not supported)	Cache-Quadrant	Cache-Hemisphere
$4K \times 4K$	0.31	0.29	—	0.30	0.32
$8K \times 8K$	1.62	1.36	—	**1.17**	1.21
$16K \times 16K$	11.71	11.54	—	**8.15**	8.61

Table 5: Run time (in seconds) of the 3-level matrix multiplication algorithm

Matrix Dimensions	Flat-SNC4	Flat-SNC2	Flat-All2All	Flat-Quadrant	Flat-Hemisphere
$32K \times 32K$	151.12	119.23	80.62	70.50	70.28
$64K \times 64K$	10540.52	884.25	768.97	720.37	653.97
Matrix Dimensions	Cache-SNC4	Cache-SNC2	Cache-All2All (not supported)	Cache-Quadrant	Cache-Hemisphere
$32K \times 32K$	93.53	94.26	—	**64.98**	68.35
$64K \times 64K$	729.14	805.44	—	686.36	**651.94**

Table 6: Run time (in seconds) of the 4-level matrix multiplication algorithm

Matrix Dimensions	Flat-SNC4	Flat-SNC2	Flat-All2All	Flat-Quadrant	Flat-Hemisphere
$64K \times 64K$	1292.55	945.01	640.35	560.46	557.92
Matrix Dimensions	Cache-SNC4	Cache-SNC2	Cache-All2All (not supported)	Cache-Quadrant	Cache-Hemisphere
$64K \times 64K$	744.34	745.16	—	**517.99**	544.31

Table 7: Run time (in seconds) of the 1-level triangular matrix multiplication using cblas_dtrmm

Matrix Dimensions	Flat-SNC4	Flat-SNC2	Flat-All2All	Flat-Quadrant	Flat-Hemisphere
$4K \times 4K$	0.46	0.40	**0.34**	0.36	0.41
$8K \times 8K$	2.43	1.50	1.31	1.41	**1.12**
$16K \times 16K$	10.97	10.38	7.83	9.12	6.21
$32K \times 32K$	73.78	79.32	62.85	71.24	62.82
$64K \times 64K$	556.95	588.33	503.83	556.77	501.58
Matrix Dimensions	Cache-SNC4	Cache-SNC2	Cache-All2All (not supported)	Cache-Quadrant	Cache-Hemisphere
$4K \times 4K$	0.49	0.38	—	0.37	0.38
$8K \times 8K$	2.04	1.43	—	1.61	**1.12**
$16K \times 16K$	7.99	7.93	—	15.40	**6.20**
$32K \times 32K$	54.03	59.04	—	77.93	**50.15**
$64K \times 64K$	450.17	445.05	—	596.00	**389.26**

Table 8: Run time (in seconds) of the 2-level triangular matrix multiplication algorithm

Matrix Dimensions	Flat-SNC4	Flat-SNC2	Flat-All2All	Flat-Quadrant	Flat-Hemisphere
$4K \times 4K$	0.44	0.40	0.35	**0.33**	0.35
$8K \times 8K$	1.65	1.32	1.16	1.33	**1.04**
$16K \times 16K$	10.02	9.52	6.91	7.72	6.02
$32K \times 32K$	69.81	59.07	52.05	51.58	51.64
$64K \times 64K$	868.32	566.61	382.54	380.42	380.55
Matrix Dimensions	Cache-SNC4	Cache-SNC2	Cache-All2All (not supported)	Cache-Quadrant	Cache-Hemisphere
$4K \times 4K$	0.35	0.35	—	**0.33**	0.32
$8K \times 8K$	1.11	1.13	—	1.26	**1.04**
$16K \times 16K$	6.87	6.60	—	7.64	**5.97**
$32K \times 32K$	50.60	50.57	—	49.77	**42.98**
$64K \times 64K$	402.92	414.56	—	422.38	**356.40**

triangular blocks is $b1 = 2$ and for ordinary blocks is $b2 = 1$. Moreover, for square-rectangle block multiplication, we called MXM_2 instead of cblas_dgemm. In the MXM_2 algorithm we used, the grids imposed on the matrices were 4×4, 4×8 and 4×8 for A, B and C respectively.

5.3 Symmetric Matrix Multiplication Times

Table 9 shows the run time values for the one-level symmetric matrix-matrix multiplication code (cblas_dsymm). Similar to what we did for triangular matrix multiplication, here again we can use this table along with Table 3 to make sure that there is the potential for performance improvement. Consider the multiplication time for $64K \times 64K$ matrices as an example. We can break matrices into $32K \times 32K$ blocks and perform 8 block multiplications 4 of which are symmetric matrix multiplication and the remaining 4 are ordinary matrix multiplication. For all memory-clustering configurations (except for flat-snc2) the time to sequentially perform four ordinary $32K \times 32K$ multiplication plus the time to sequentially perform four symmetric $32K \times 32K$ multiplications is less than the time to perform one symmetric $64K \times 64K$ multiplication. As a result, the initial test has passed and the potential for speed up exists for a simple sequential model using 2×2 grids to block the matrices. We can also use parallelism and different settings for block dimensions.

Table 10 shows the run time values for our two-level algorithm. In this implementation, matrix A is divided using a 2×2 grid and matrices B and C are divided using a 2×1 grid each. The degree of concurrency for both symmetric and ordinary blocks is $b1 = b2 = 2$.

5.4 Run Time Comparison For Different Configuration Selection Scenarios

Throughout this section, we use the layout convention given in Table 11 when we refer to "our algorithm". For example, the 3-level algorithm for matrix multiplication is the same one for which we reported runtimes in Section 5.1 and has the same parameters.

Figures 3, 4 and 5 compare the algorithms under scenario S1 (each algorithm uses the best configuration for each matrix size), for matrix multiplication, triangular matrix multiplication and symmetric matrix multiplication respectively. For matrix multiplication, our algorithm results in 20.5% speed up for $16K \times 16K$ matrices, 19.0% for $32K \times 32K$ matrices and 16.2% for $64K \times 64K$ matrices. The corresponding speed up values are 3.7%, 14.3% and 8.4% for triangular matrix multiplication and -0.5%, 11.5% and 10.5% for symmetric matrix multiplication.

Tables 12, 13 and 14 show the comparison results using scenario S2 (all algorithms use the same configuration) for matrix multiplication, triangular matrix multiplication and symmetric matrix multiplication respectively. The comparisons are expressed as the percentage runtime reduction obtained by our algorithm over the 1-level code for each of the above-mentioned applications. In all the three tables, the negative percentages (where our algorithm has performed worse than the 1-level code) appear at the configurations involving snc2 or snc4 (Table 14 is exception, but the negative speed up values we get for other configurations have very small absolute values). We will briefly explain in Section 5.5 why our method performs poorly in the Sub-NUMA Clustering modes. If we ignore the modes involving snc2 and snc4, then for matrix multiplication on $16K \times 16K$ matrices, the speed up we get is in range [13.7%, 33.8%]; for $32K \times 32K$ matrices it is in range [14.8%, 33.4%] and for $64K \times 64K$ matrices it is in range [16.6%, 40.5%]. The corresponding ranges for triangular matrix multiplication are [3.1%, 50.4%], [14.3%, 36.1%] and [8.4%, 31.7%] and for symmetric matrix multiplication are [−0.9%, 50.9%], [10.3%, 28.2%] and [11.6%, 30.1%].

For the third scenario S3, we make the following observations ($conf_1$ is the best configuration for the 1-level algorithm and $conf_m$ is the best one for our multilevel algorithm):

- For matrix multiplication, $conf_1$: cache-hemisphere and $conf_m$: cache-quadrant (for $conf_m$, results of 2-level, 3-level and 4-level algorithms have been considered collectively).
- For triangular matrix multiplication, $conf_1$: cache-hemisphere and $conf_m$: cache-hemisphere.
- For symmetric matrix multiplication, $conf_1$: cache-hemisphere and $conf_m$: cache-hemisphere.

Figures 6, 7 and 8 compare our algorithms vs. the 1-level code for scenario S3. Using our speed up method on matrix multiplication, we get 21.3% speed up for $16K \times 16K$ matrices, 19.0% for $32K \times 32K$ matrices and 20.6% for $64K \times 64K$ matrices. The corresponding speed up values are 3.7%, 14.3% and 8.4% for triangular matrix multiplication and -0.9%, 10.3% and 11.6% for symmetric

Table 9: Run time (in seconds) of the 1-level symmetric matrix multiplication algorithm using cblas_dsymm

Matrix Dimensions	Flat-SNC4	Flat-SNC2	Flat-All2All	Flat-Quadrant	Flat-Hemisphere
$4K \times 4K$	0.67	0.57	**0.47**	0.54	0.56
$8K \times 8K$	4.47	2.63	2.33	2.97	1.99
$16K \times 16K$	24.78	20.75	15.12	32.13	**11.96**
$32K \times 32K$	130.27	152.23	122.94	120.16	123.49
$64K \times 64K$	1006.21	957.23	994.72	1002.23	1005.31
Matrix Dimensions	Cache-SNC4	Cache-SNC2	Cache-All2All (not supported)	Cache-Quadrant	Cache-Hemisphere
$4K \times 4K$	0.67	0.55	—	0.53	0.55
$8K \times 8K$	2.98	2.50	—	2.87	**1.97**
$16K \times 16K$	13.43	13.98	—	31.66	12.00
$32K \times 32K$	99.14	109.71	—	153.31	**98.17**
$64K \times 64K$	791.85	**778.54**	—	1346.62	797.71

Table 10: Run time (in seconds) of 2-level symmetric matrix multiplication algorithm

Matrix Dimensions	Flat-SNC4	Flat-SNC2	Flat-All2All	Flat-Quadrant	Flat-Hemisphere
$4K \times 4K$	0.75	0.51	0.41	0.39	0.39
$8K \times 8K$	3.59	2.59	2.15	2.27	1.80
$16K \times 16K$	24.09	19.95	14.78	15.78	**12.02**
$32K \times 32K$	93.36	164.51	89.49	89.71	90.72
$64K \times 64K$	1631.96	1312.09	711.02	711.50	709.60
Matrix Dimensions	Cache-SNC4	Cache-SNC2	Cache-All2All (not supported)	Cache-Quadrant	Cache-Hemisphere
$4K \times 4K$	0.45	0.45	—	**0.36**	0.38
$8K \times 8K$	2.08	2.16	—	2.19	**1.77**
$16K \times 16K$	12.85	13.41	—	16.05	12.11
$32K \times 32K$	94.60	100.01	—	110.11	**88.03**
$64K \times 64K$	826.87	765.08	—	941.59	**704.79**

matrix multiplication. Note that for the latter two applications as $conf_1 = conf_m$, the speed up values are the same as the ones in the "cache-hemisphere" column of Tables 13 and 14 respectively.

Table 11: The algorithm we use (number of levels) depends on the target application and the size of the input matrices.

	$16K \times 16K$ matrices	$32K \times 32K$ matrices	$64K \times 64K$ matrices
Matrix Multiplication (MXM)	2-level algorithm	3-level algorithm	4-level algorithm
Triangular MXM	2-level algorithm		
Symmetric MXM	2-level algorithm		

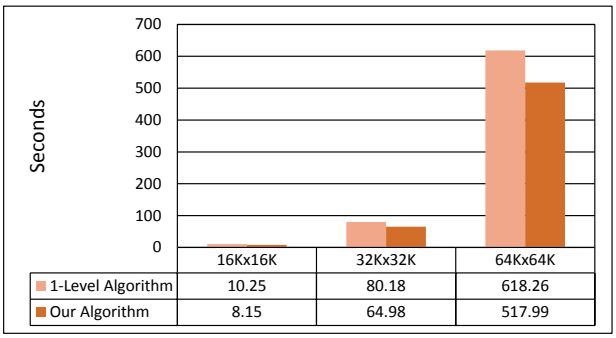

Figure 3: Comparison of our method against the 1-level matrix multiplication algorithm (cblas_dgemm) using scenario S1.

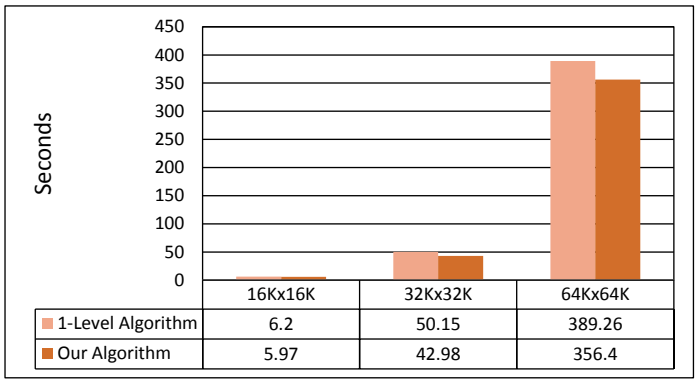

Figure 4: Comparison of our method against the 1-level triangular matrix multiplication algorithm (cblas_dtrmm) using scenario S1.

5.5 *Model Validation*

In this section, we validate the mathematical model we developed in Section 3.5 to predict performance improvement. The validation is done on matrix multiplication. It is similar for triangular and symmetric matrix multiplication problems. We show that the $rbk > 1$ condition correctly predicts the potential for speed up. Also, we compare the actual $\frac{T_i}{T_1}$ ratios with those predicted using our formulation. For predicting the ratios, we use equations (5) and (8) ignoring the $g(.,.)$ terms. We ignore the $g(.,.)$ terms by assuming

that $g(c,n) \ll T_1$ (the same assumption as the one we made several times in Section 3.5). Meaning that we use (33) and (34) to predict the runtime ratios.

$$\frac{T_2}{T_1} = \frac{f(c)}{b} \times \frac{T^1_{p,n}}{T_1} \tag{33}$$

$$\frac{T_i}{T_1} = \frac{f(c_1)T^{i-1}_{bp,c_2 n}}{T_1} \tag{34}$$

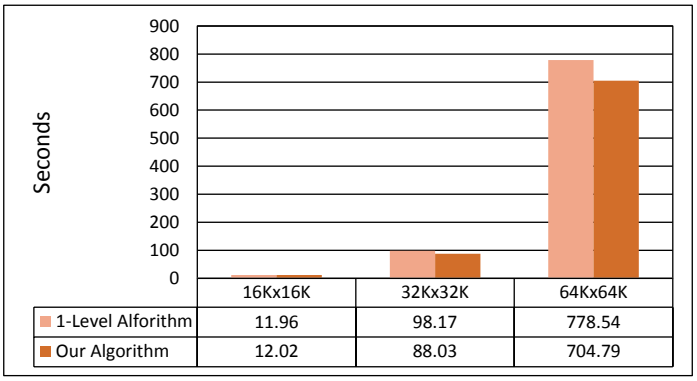

Figure 5: Comparison of our method against the 1-level symmetric matrix multiplication algorithm (cblas_dsymm) using scenario S1.

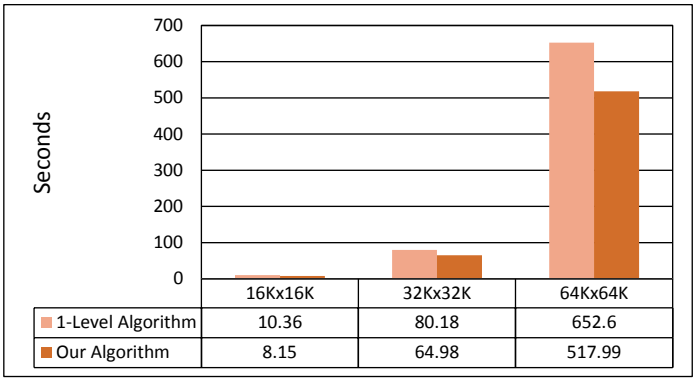

Figure 6: Comparison of our method against the 1-level matrix multiplication algorithm (cblas_dgemm) for scenario S3.

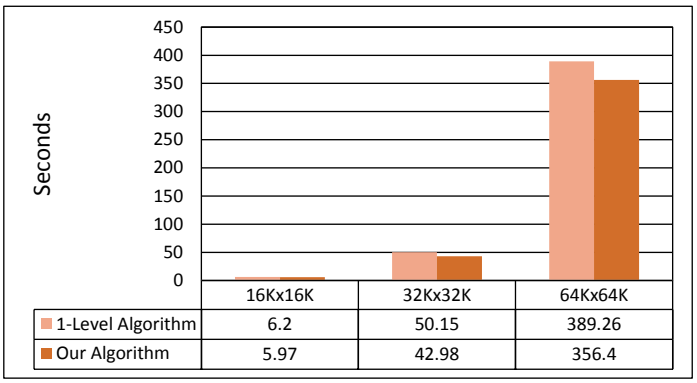

Figure 7: Comparison of our method against the 1-level triangular matrix multiplication algorithm (cblas_dtrmm) for scenario S3.

In our analysis we ignore matrix addition time as well as the time spent on copying the matrix blocks into the fast memory. The reason is that theoretically those are $O(n^2)$ operations while matrix multiplication is an $O(n^3)$ operation. In practice, for our two - level matrix multiplication code in the default KNL mode (cache - quadrant) for instance, the ratio (multiplication time)/(data transfer time) is 3 for matrices of size $4K \times 4K$ and 42 for matrices of size $16K \times 16K$. The ratio (multiplication time)/(addition time) is 21 and 199, for matrix sizes of $4K \times 4K$ and $16K \times 16K$ respectively. Similar ratios can be obtained for other matrix sizes and using other configurations.

Table 15 shows the validation results for our T_2 formula in Equation 5. Here, $cn = 16K$, number of blocks $c = 4$ and therefore $n = 4K$. The degree of concurrency b is 4. Here and also in the following validation experiments, the total number of processors $bp = 64$. The error between the computed value of T_2/T_1 and the measured value ranges from 1.27% (Flat-all2all) to 19.44% (Cache-snc4) in all the modes except for Flat-snc4 (86.09%).

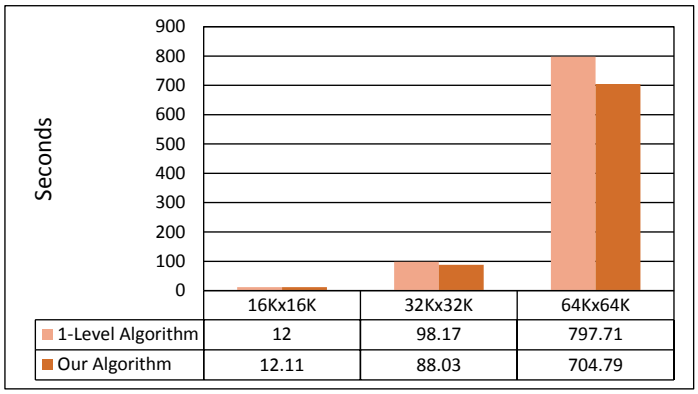

Figure 8: Comparison of our method against the 1-level symmetric matrix multiplication algorithm (cblas_dsymm) for scenario S3.

Table 16 shows the validation results for our T_3 formula. Here, matrix size $cn = 32K$, number of blocks in the first (outermost) level $c_1 = 2$ (arranged as a 2×2 grid) and therefore the outermost blocks are of size $16K \times 16K$. The error between computed T_3/T_1 and the observed value ranges from 0% (Flat-snc2, Flat-all2all) to 1.85% (Cache-snc2) for all the modes except for Flat-snc4 where it is 502%.

Table 17 shows the validation results for our T_4 formula. Here, matrix size $cn = 64K$, number of blocks in the first (outermost) level $c_1 = 2$ (arranged as a 2×2 grid) and therefore the outermost blocks are $32K \times 32K$. Note that in tables 15, 16 and 17 the computed and the actual runtime ratios have been rounded to the nearest hundredth and that the error values are computed based on those rounded ratios. Here, the error between the computed and the observed values ranges from 0% (Flat - quadrant) to 1.69% (Cache - quadrant) in all the modes except for the Flat-snc4 mode where it is 6.2%.

Figures 9, 10 and 11 visually show the actual values versus the computed values for T_3/T_1, T_2/T_1 and T_4/T_1 respectively.

Our validation results show that our model works fairly well for all the KNL configurations except for those involving snc2 or snc4. The reason is that in the Sub-NUMA Clustering modes, each half (or quadrant) of the chip is exposed as a separate domain to the OS, essentially making the chip analogous to a 2 or 4 socket Xeon. This restricts data sharing and memory access time will increase as a result.

The condition for speed up, namely $rbk > 1$ works correctly for 8 out of 9 KNL configurations (the only exception being Flat-snc4). Note that the *rbk* column is the same in all the three tables. That is because its value only depends on the parameters of levels 1 and 2.

Table 12: The speed up obtained by our algorithm over the 1-level matrix multiplication code (cblas_dgemm) for scenario S2,

Matrix Dimensions	Flat-SNC4	Flat-SNC2	Flat-All2All	Flat-Quadrant	Flat-Hemisphere
$16K \times 16K$	-389.0%	3.9%	20.6%	25.6%	13.7%
$32K \times 32K$	-26.1%	1.8%	21.4%	30.6%	31.5%
$64K \times 64K$	-29.0%	-12.1%	24.0%	32.2%	32.3%
Matrix Dimensions	Cache-SNC4	Cache-SNC2	Cache-All2All (not supported)	Cache-Quadrant	Cache-Hemisphere
$16K \times 16K$	-7.5%	-1.9%	—	**33.8%**	16.9%
$32K \times 32K$	-5.2%	-7.9%	—	**33.4%**	14.8%
$64K \times 64K$	-20.4%	4.7%	—	**40.5%**	16.6%

Table 13: The speed up obtained by our algorithm over the 1-level triangular matrix multiplication code (cblas_dtrmm) for Scenario S2.

Matrix Dimensions	Flat-SNC4	Flat-SNC2	Flat-All2All	Flat-Quadrant	Flat-Hemisphere
$16K \times 16K$	8.7%	8.3%	11.7%	15.4%	3.1%
$32K \times 32K$	5.4%	25.5%	17.2%	27.6%	17.8%
$64K \times 64K$	-55.9%	3.7%	24.1%	**31.7%**	24.1%
Matrix Dimensions	Cache-SNC4	Cache-SNC2	Cache-All2All (not supported)	Cache-Quadrant	Cache-Hemisphere
$16K \times 16K$	14.0%	16.8%	—	**50.4%**	3.7%
$32K \times 32K$	6.3%	14.3%	—	**36.1%**	14.3%
$64K \times 64K$	10.5%	6.9%	—	29.1%	8.4%

Table 14: The speed up obtained by our algorithm over the 1-level symmetric matrix multiplication code (cblas_dsymm) for Scenario S2.

Matrix Dimensions	Flat-SNC4	Flat-SNC2	Flat-All2All	Flat-Quadrant	Flat-Hemisphere
$16K \times 16K$	2.8%	3.9%	2.2%	**50.9%**	-0.5%
$32K \times 32K$	**28.3%**	-8.1%	27.2%	25.3%	26.5%
$64K \times 64K$	-62.2%	-37.1%	28.5%	29.0%	29.4%
Matrix Dimensions	Cache-SNC4	Cache-SNC2	Cache-All2All (not supported)	Cache-Quadrant	Cache-Hemisphere
$16K \times 16K$	4.3%	4.1%	—	49.3%	-0.9%
$32K \times 32K$	4.6%	8.8%	—	**28.2%**	10.3%
$64K \times 64K$	-4.4%	1.7%	—	**30.1%**	11.6%

Table 15: Evaluation of T_2 formula against different KNL configurations - All time values are in seconds

	T_1	T_2	$T^1_{p,n}$	$T^1_{bp,n}$	$T^1_{p,cn}$	Computed T_2/T_1	Actual T_2/T_1	Error	*rbk*
Flat - SNC4	23.21	113.51	0.98	0.65	46.04	0.68	4.89	86.09%	1.48
Flat - SNC2	15.42	14.82	0.84	0.39	51.67	0.87	0.96	9.37%	1.15
Flat - All2All	12.74	10.12	0.64	0.3	52.88	0.80	0.79	1.27%	1.24
Flat - Quadrant	11.92	8.87	0.57	0.33	35.62	0.77	0.74	4.05%	1.31
Flat - Hemisphere	10.25	8.85	0.59	0.3	38.02	0.92	0.86	6.98%	1.09
Cache - SNC4	10.89	11.71	0.88	0.48	38.03	1.29	1.07	19.44%	0.77
Cache - SNC2	11.32	11.54	0.81	0.42	42.56	1.14	1.02	11.76%	0.87
Cache - Quadrant	12.32	8.15	0.55	0.32	35.79	0.71	0.66	7.58%	1.40
Cache - Hemisphere	10.36	8.61	0.56	0.29	38.33	0.86	0.83	3.61%	1.16

Table 16: Evaluation of T_3 formula against different configurations - All time values are in seconds

	T_1	T^2_{bp,c_2n}	T_3	Computed T_3/T_1	Actual T_3/T_1	Error	*rbk*
Flat - SNC4	119.81	113.51	151.12	7.58	1.26	501.59%	1.48
Flat - SNC2	121.47	14.82	119.23	0.98	0.98	0.00%	1.15
Flat - All2All	102.53	10.12	80.62	0.79	0.79	0.00%	1.24
Flat - Quadrant	101.58	8.87	70.5	0.70	0.69	1.45%	1.31
Flat - Hemisphere	102.63	8.85	70.28	0.69	0.68	1.47%	1.09
Cache - SNC4	88.9	11.71	93.53	1.05	1.05	0.00%	0.77
Cache - SNC2	87.39	11.54	94.26	1.06	1.08	1.85%	0.87
Cache - Quadrant	97.52	8.15	64.98	0.67	0.67	0.00%	1.40
Cache - Hemisphere	80.18	8.61	68.35	0.86	0.85	1.18%	1.16

Table 17: Evaluation of T_4 formula against different KNL configurations - All time values are in seconds

	T_1	T^3_{bp,c_2n}	T_4	Computed T_4/T_1	Actual T_4/T_1	Error	*rbk*
Flat - SNC4	1002.3	151.12	1292.55	1.21	1.29	6.20%	1.48
Flat - SNC2	843.25	119.23	945.01	1.13	1.12	0.89%	1.15
Flat - All2All	843.04	80.62	640.35	0.77	0.76	1.32%	1.24
Flat - Quadrant	826.23	70.5	560.46	0.68	0.68	0.00%	1.31
Flat - Hemisphere	824.15	70.28	557.92	0.68	0.68	0.00%	1.09
Cache - SNC4	618.26	93.53	744.34	1.21	1.20	0.83%	0.77
Cache - SNC2	781.83	94.26	745.16	0.96	0.95	1.05%	0.87
Cache - Quadrant	871.1	64.98	517.99	0.60	0.59	1.69%	1.40
Cache - Hemisphere	652.6	68.35	544.31	0.84	0.83	1.20%	1.16

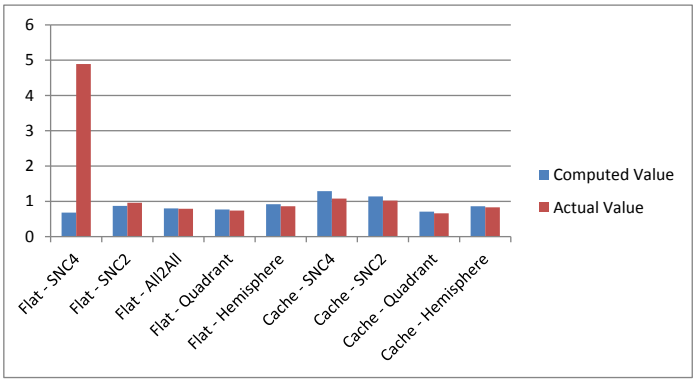

Figure 9: T_2/T_1: Computed Values vs. Actual Values. Demonstrated over KNL memory-clustering configurations.

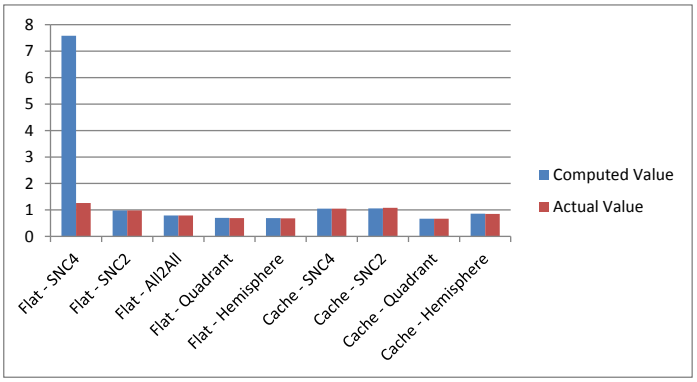

Figure 10: T_3/T_1: Computed Values vs. Actual Values. Demonstrated over KNL memory-clustering configurations.

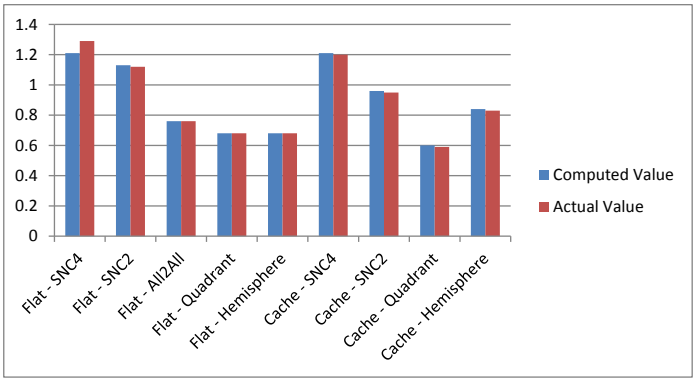

Figure 11: T_4/T_1: Computed Values vs. Actual Values. Demonstrated over KNL memory-clustering configurations.

If we leave out the Flat-snc4 configuration, then the speed up predictions of our model (either negative or positive) are within 2%, 2% and 20% of the actual values for matrix sizes $64K \times 64K$, $32K \times 32K$ and $16K \times 16K$, respectively.

6 Conclusions

We have proposed a multilevel method to speed parallel codes whose runtime grows faster than their workload. The target problems are the ones that can be decomposed into smaller subproblems. The original highly optimized parallel code is treated as a black box and no change is made to it. Using a simple parallel computing model, we derived formulas to predict whether or not any speed up is possible and also the amount of attainable speed up.

We demonstrated the effectiveness of our multilevel method on the highly optimized parallel BLAS routines cblas_dgemm, cblas_dtrmm and cblas_dsymm that are in the MKL library using the Intel KNL platform. Our proposed method is, however, general and can be potentially applied to other optimized parallel codes and on other platforms.

In our experiments, we considered 3 possible application scenarios. In the first of these (each algorithm is run using the best memory/clustering configuration for each problem size), we obtained speed up values of 16.2%, 8.4%, and 10.5%, respectively, for matrix multiplication, triangular matrix multiplication, and symmetric matrix multiplication for $64K \times 64K$ matrices. In the second scenario (all algorithms use the same configuration), these percentages are up to 40.5%, 31.7%, and 30.1%. The default mode in the KNL is cache - quadrant configuration. For this mode, the speed up percentages are 40.5%, 29.1%, and 30.1%. In the third scenario (each algorithm uses the same configuration regardless of problem size; different algorithms may use different configurations), the speed up values were 20.6%, 8.4%, and 11.6%.

For the matrix multiplication problem, the simple predictive mathematical model developed by us is able to correctly predict whether or not speed up is possible for 8 out of the 9 memory/clustering configurations tested. Also, for 8 out of the 9 configurations and for the matrix sizes of $64K \times 64K$, the amount of runtime increase or decrease our formula predicted for matrix multiplication was within 2% of the actual value. For both predicting the potential of speed up and predicting the amount of attainable speed up, Flat-snc4 was the configuration for which our model failed.

Conflict of Interest The authors declare no conflict of interest.

Acknowledgment This work was funded, in part, by the National Science Foundation, under Contract No. 1748652.

References

[1] S. Gheibi, T. Banerjee, S. Ranka, S. Sahni, "Multilevel Approaches to Fine Tune Performance of Linear Algebra Libraries," in 2019 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT), 1–6, IEEE, 2019.

[2] L. E. Cannon, A cellular computer to implement the Kalman filter algorithm, Ph.D. thesis, Montana State University-Bozeman, College of Engineering, 1969.

[3] V. Strassen, "Gaussian elimination is not optimal," Numerische mathematik, **13**(4), 354–356, 1969.

[4] H. Prokop, Cache-oblivious algorithms, Ph.D. thesis, Massachusetts Institute of Technology, 1999.

[5] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, K. H. Randall, "An analysis of dag-consistent distributed shared-memory algorithms," in SPAA, volume 96, 297–308, 1996.

[6] J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, O. Spillinger, "Communication-optimal parallel recursive rectangular matrix multiplication," in 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, 261–272, IEEE, 2013.

[7] B. Lipshitz, G. Ballard, J. Demmel, O. Schwartz, "Communication-avoiding parallel strassen: Implementation and performance," in SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 1–11, IEEE, 2012.

[8] G. H. Golub, C. F. Van Loan, Matrix computations, volume 3, JHU press, 2012.

[9] R. A. Van De Geijn, J. Watts, "SUMMA: Scalable universal matrix multiplication algorithm," Concurrency: Practice and Experience, **9**(4), 255–274, 1997.

[10] M. M. A. Patwary, N. R. Satish, N. Sundaram, J. Park, M. J. Anderson, S. G. Vadlamudi, D. Das, S. G. Pudov, V. O. Pirogov, P. Dubey, "Parallel efficient sparse matrix-matrix multiplication on multicore platforms," in International Conference on High Performance Computing, 48–57, Springer, 2015.

[11] Q. Xiangzhen, "Cache performance and algorithm optimization," in High Performance Computing on the Information Superhighway, 1997. HPC Asia'97, 12–17, IEEE, 1997.

[12] D. I. Lyakh, "An efficient tensor transpose algorithm for multicore CPU, Intel Xeon Phi, and NVidia Tesla GPU," Computer Physics Communications, **189**, 84–91, 2015.

[13] J. Chen, J. Fang, W. Liu, T. Tang, C. Yang, "clmf: A fine-grained and portable alternating least squares algorithm for parallel matrix factorization," Future Generation Computer Systems, **108**, 1192–1205, 2020.

[14] C. Yount, A. Duran, "Effective use of large high-bandwidth memory caches in HPC stencil computation via temporal wave-front tiling," in Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), International Workshop on, 65–75, IEEE, 2016.

[15] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, M. Thottethodi, "Nonlinear array layouts for hierarchical memory systems," in Proceedings of the 13th international conference on Supercomputing, 444–453, ACM, 1999.

[16] J. Mellor-Crummey, D. Whalley, K. Kennedy, "Improving memory hierarchy performance for irregular applications using data and computation reorderings," International Journal of Parallel Programming, **29**(3), 217–247, 2001.

[17] E. Athanasaki, N. Koziris, "Fast indexing for blocked array layouts to improve multi-level cache locality," in Interaction between Compilers and Computer Architectures, 2004. INTERACT-8 2004. Eighth Workshop on, 107–119, IEEE, 2004.

[18] C. Kulkarni, C. Ghez, M. Miranda, F. Catthoor, H. De Man, "Cache conscious data layout organization for embedded multimedia applications," in Proceedings of the conference on Design, automation and test in Europe, 686–693, IEEE Press, 2001.

[19] B. Recht, C. Re, S. Wright, F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in Advances in neural information processing systems, 693–701, 2011.

[20] W.-S. Chin, Y. Zhuang, Y.-C. Juan, C.-J. Lin, "A fast parallel stochastic gradient method for matrix factorization in shared memory systems," ACM Transactions on Intelligent Systems and Technology (TIST), **6**(1), 2, 2015.

[21] S. Song, J. K. Hollingsworth, "Designing and auto-tuning parallel 3-D FFT for computation-communication overlap," in Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming, 181–192, 2014.

[22] S. Lee, D. Jha, A. Agrawal, A. Choudhary, W.-k. Liao, "Parallel deep convolutional neural network training by exploiting the overlapping of computation and communication," in 2017 IEEE 24th International Conference on High Performance Computing (HiPC), 183–192, IEEE, 2017.

[23] H. Wang, S. Guo, R. Li, "Osp: Overlapping computation and communication in parameter server for fast machine learning," in Proceedings of the 48th International Conference on Parallel Processing, 1–10, 2019.

[24] J. Huang, T. M. Smith, G. M. Henry, R. A. van de Geijn, "Strassen's algorithm reloaded," in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 59, IEEE Press, 2016.

[25] R.-I. Ciobanu, C. Dobre, M. Bălănescu, G. Suciu, "Data and task offloading in collaborative mobile fog-based networks," IEEE Access, **7**, 104405–104422, 2019.

[26] V. Priya, C. S. Kumar, R. Kannan, "Resource scheduling algorithm with load balancing for cloud service provisioning," Applied Soft Computing, **76**, 416–424, 2019.

[27] D. Puthal, R. Ranjan, A. Nanda, P. Nanda, P. P. Jayaraman, A. Y. Zomaya, "Secure authentication and load balancing of distributed edge datacenters," Journal of Parallel and Distributed Computing, **124**, 60–69, 2019.

[28] A. Mallick, M. Chaudhari, U. Sheth, G. Palanikumar, G. Joshi, "Rateless codes for near-perfect load balancing in distributed matrix-vector multiplication," Proceedings of the ACM on Measurement and Analysis of Computing Systems, **3**(3), 1–40, 2019.

[29] A. Mohammed, A. Eleliemy, F. M. Ciorba, F. Kasielke, I. Banicescu, "An approach for realistically simulating the performance of scientific applications on high performance computing systems," Future Generation Computer Systems, **111**, 617–633, 2020.

[30] I.-J. Sung, J. A. Stratton, W.-M. W. Hwu, "Data layout transformation exploiting memory-level parallelism in structured grid many-core applications," in Proceedings of the 19th international conference on Parallel architectures and compilation techniques, 513–522, ACM, 2010.

[31] G. Chen, B. Wu, D. Li, X. Shen, "PORPLE: An extensible optimizer for portable data placement on GPU," in Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, 88–100, IEEE Computer Society, 2014.

[32] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, Y.-C. Liu, "Knights landing: Second-generation intel xeon phi product," Ieee micro, **36**(2), 34–46, 2016.