Neural, Parallel, and Scientific Computations 28(2020), No.3, 198 - 222

Exploiting the Capabilities of Distributed Multi-Core Intel Processors for Accelerating Dense Linear Algebra

FATMA S. AHMED AND MOSTAFA I. SOLIMAN
Computer and System Section, Electrical Engineering Department, Faculty of Engineering, Aswan University, Aswan 81542, Egypt.
fatma.sayed@aswu.edu.eg and mossol@ieee.org/mossol@yahoo.com

Abstract. This paper exploits the capabilities of distributed multi-core Intel processors for accelerating dense linear algebra used in most calculations of scientific computing. Some kernels from BLAS (applying Givens rotation, rank-1 update, and matrix multiplication) and SVD are implemented and evaluated on the target system (cluster of Fujitsu Siemens CELSIUS R550 multi-core Intel processors). On a quad-core Intel Xeon E5410 processor running at 2.33 GHz, the maximum performance of applying Givens rotation (Level-1 BLAS) is improved from 2.10 to 8.08, 3.00, and 3.64 GFLOPS using SIMD, multi-threading, and multi-threading SIMD techniques, respectively. However, the use of MPI on multiple nodes degrades the performance because the network overhead for sending/receiving data/results dominates the overall execution time. For the same reason, the performance of rank-1 update (Level-2 BLAS) due to using multi-threading, SIMD, and blocking techniques degrades from 2.33 to 4.37×10^{-2} GFLOPS when eight nodes are used for parallel processing. The speedups of the traditional matrix-matrix multiplication (Level-3 BLAS) on a single quad-core Intel Xeon E5410 processor over the sequential execution when applying SIMD, multi-threading, multi-threading SIMD, and multi-threading SIMD blocking techniques are 3.76, 3.91, 7.37, and 12.65, respectively. Moreover, on ten nodes, the performance of traditional matrix-matrix multiplication reaches 99.73 GFLOPS. Finally, the executions of block Jacobi and hierarchal block Jacobi on eight nodes with applying SIMD and multi-threading techniques give performances of 206.97 and 515.62 GFLOPS, respectively. The speedups over sequential one-sided Jacobi are 49.8 and 124, respectively.

Keywords – ILP/DLP/TLP; MPI; SVD; SIMD; multi-threading.

1. INTRODUCTION

Nowadays, it is widely accepted that exploiting all forms of parallelism is the only way to significantly improve the performance. The three major forms of parallelism on a modern processor are (1) instruction-level parallelism (ILP), (2) data-level parallelism

Received August 20, 2020
www.dynamicpublishers.org;
https://doi.org/10.46719/npsc20202834

(DLP), and (3) thread-level parallelism (TLP), which are not mutually exclusive [1]. Therefore, the use of message passing interface (MPI) on a cluster of multi-core processors can improve the performance of applications by exploiting ILP, DLP, and TLP on a distributed system. This paper shows how the dense linear algebra used in most calculations of scientific computing is accelerated by exploiting the capabilities of distributed multi-core Intel processors.

Some kernels of dense linear algebra are implemented and evaluated on a cluster of Fujitsu Siemens CELSIUS R550 [2]. In such cluster, each computer has quad-core Intel Xeon E5410 processor running at 2.33 GHz, L1 data cache of 32 KB and L1 instruction cache of 32 KB for each core, shared L2 cache of 12 MB, and 4 GB of main memory, as illustrated in Figure 1 [3]. Moreover, the target processor includes data prefetch logic and thirty functional execution units in eleven groups (six general-purpose ALUs, two integer units, one shift unit, four data cache units, six multimedia units, two parallel shift units, one parallel multiply, two 82-bit floating-point multiply-accumulate units, two SIMD floating-point multiply-accumulate units, and three branch units).

The Intel Xeon E5410 processor supports streaming SIMD extensions SSE, SSE2, SSE3, SSE3S, and SSE4.1, see [3] for more details. The floating-point and multimedia units include sixteen 128-bit wide registers (XMM0 – XMM15) and a separate register for data movement. It supports Intel-64 architecture, and includes compatibility with IA-32 software. The base data word is 64 bits and byte-addressable memory. The logical address space is 2^{64} bytes. Intel Xeon E5410 uses a hardware register renaming technique, which is one of the ways for exploiting ILP [4]. The same technique is also used to permit parallel execution of loops. The architecture implements eight branch registers. The fetch technique can read up to two instruction words per clock from the L1 cache into the pipeline. It includes coarse multithreading hardware by which each processor core maintains context for two threads of execution [5]. When one thread stalls during memory access, the other thread can execute. From a software point of view, each computer in the cluster is running

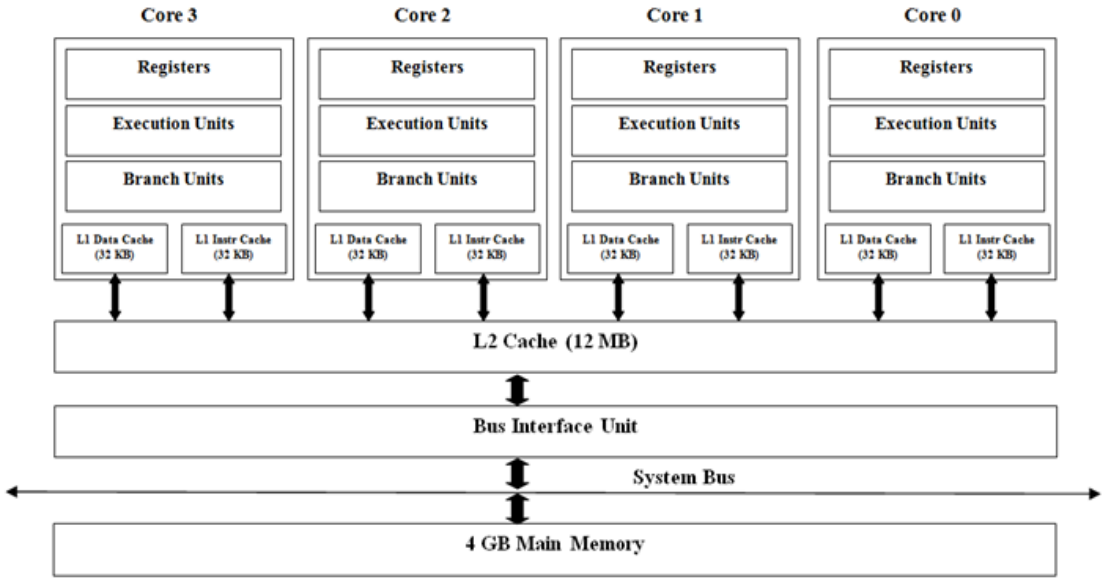


Figure 1: Quad-core Intel Xeon E5410 processor for the target system.

Microsoft Window 7 operating system, where Microsoft visual studio 2012 and MPICH2-1.2.1p1 library are used for MPI programming.

Some kernels from BLAS (basic linear algebra subprograms) and SVD (singular value decomposition as a representative example of dense matrix factorization) are implemented and evaluated on the target system (cluster of Fujitsu Siemens CELSIUS R550 multi-core Intel processors). Apply Givens rotation, rank-1 update, and matrix multiplication are selected from the Level-1 (vector-vector operations) [6], Level-2 (matrix-vector operations) [7], and Level-3 (matrix-matrix operations) BLAS [8], respectively. Applying Givens rotation is the process of replacement elements of two vectors *x* and *y* with length *n* to new values according to the following equation:

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \begin{bmatrix} x_i \\ y_i \end{bmatrix}, \text{ for } i = 0, 1, 2, 3, \ldots \ldots, \text{ n-1}$$

where $x_i = c \times x_i - s \times y_i$ and $y_i = s \times x_i + c \times y_i$. Note that *c* and *s* are the parameters of the Givens rotation, which can be calculated by using construct Givens rotation subroutine. This subroutine has 6*n* floating-point operations (FLOPs) and performs 2*n* load/store operations, where *n* is the vector length. The ratio of FLOPs to memory references is 3.

The Rank-1 update is the operation of updating the elements of an $n \times n$ matrix A by multiplying two *n* vectors *x* and *y*. The semantic rank-1 update is:

$$A_{(i,j)} = A_{(i,j)} + x_i y_j, \ where\ 0 \leq i, j < n.$$

The programming of rank-1 update has two nested loops (*i* and *j*), which results in two variants (*ij* and *ji*). These variants take $2n^2$ FLOPs and have the same number of memory references ($3n^2 + n$). For either *ij* or *ji* variant, n^2, n^2, *n* and n^2 are for loading matrix *A*, loading vector *y* or *x* by *n* times, loading vector *x* or *y*, and storing matrix *A*. Although the two variants have the same FLOPs and memory references, the performance of the *ij* variant is better than that of the *ji* variant. This difference in the performance because of their access patterns of memory are different, where *ij* access the matrix *A* by row (one stride), however, by column (*n* stride) in *ji* variant. The way in which an array's elements are referenced is called a stride; it is equal to the difference of the addresses of successive elements over the element size.

Matrix-matrix multiplication is a way to combine two matrices and get a third matrix: $C_{n \times m} = A_{n \times p} \times B_{p \times m}$. As the matrix-matrix multiplication is implemented by using triply nested loops (*i*, *j*, and *k*), there are six variants (*ijk*, *ikj*, *jik*, *jki*, *kij*, and *kji*) for the multiplication [9]. These six variants differ in their access patterns of memory; however, they have the same number of FLOPs ($2n^3$). These variants can be calculated by interchanging the order of *i*, *j*, and *k* loops. The best variant of the conventional matrix product is *ikj* because this variant accesses the memory by a unit stride. As all the elements in a loaded cache line are used, the use of unit stride leads to higher performance. As the *ikj* variant is the best one, it is used on all implementations of SIMD, multi-threading, and MPI discussed in the next sections.

In linear algebra, singular value decomposition (SVD) is an important factorization of a real matrix. It is used in many applications such as signal processing, data mining,

statistics, etc. [10-11]. SVD problem is a very computationally intensive problem that needs to exploit the growing availability of parallel hardware. The factorization of a real matrix $A_{m\times n}$ ($m \geq n$) into the product of three matrices is called SVD as follows:

$$A_{m\times n} = U_{m\times m}\sum_{m\times n}V^{T}{}_{n\times n},$$

where, $U_{m\times m}$ and $V_{n\times n}$ are orthogonal matrices (i.e., $U^{T}U = I_m$ and $V^{T}V = I_n$), which contain m left and n right singular vectors respectively, and $\sum_{m\times n}$ is a diagonal matrix diag (σ_1, σ_2, σ_3,…, σ_n) on the top of $m - n$ rows of zeros. There are relationships between the singular values of $A_{m\times n}$ (σ_i) and its singular vectors (u_i and v_i) as follows:

$$Av_i = \sigma_i u_i \text{ and } A^{T}u_i = \sigma_i v_i.$$

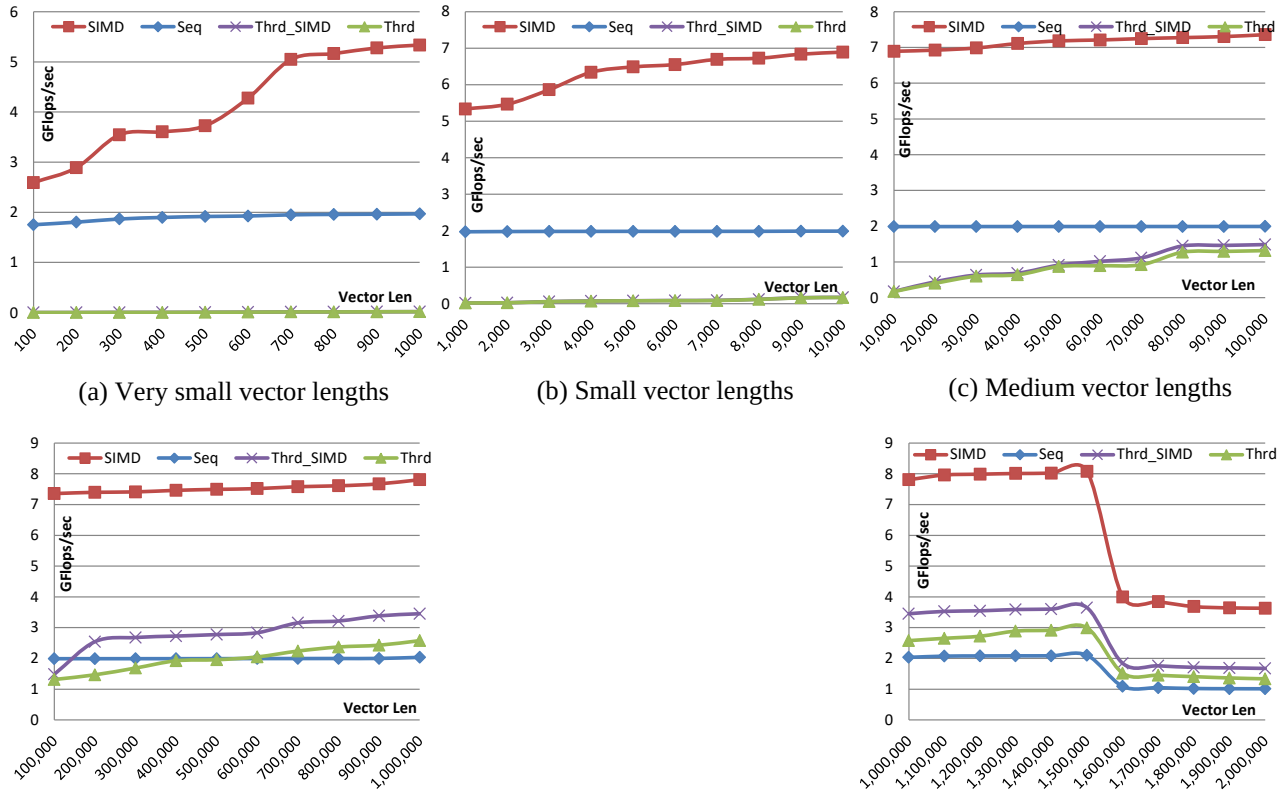
Many researchers have worked on designing efficient techniques to compute SVDs on parallel to reduce the execution time, especially for real time applications [12, 13]. Soliman [14] presented a block Jacobi algorithm for computing SVD on multiple processors. This algorithm partitions the rows of the input matrix into *2P* blocks (panels of rows), where *P* is the parallel processing cores. Each core computes the computation of two blocks. On large matrix sizes, the performance of the block Jacobi algorithm decreases, because of increasing the rate of cache misses. For computers with memory hierarchy, it is preferable to perform the computation on blocks of data instead of vectors to reduce the impact of memory latency by reusing the loaded data in cache memories. Moreover, Soliman [15] proposed a new algorithm called hierarchal block Jacobi (HBJ) for parallel computing SVD on multi-level memory hierarchy architectures by restructuring the well-known one-sided Jacobi method. HBJ partitions the given matrix into super-rows (panels of rows) to exploit the memory hierarchy by performing all computations on super-rows instead of on rows. Each super-row consists of a set of consecutive rows of the input matrix. To compute HBJ on *P* parallel processing cores, these super-rows are partitioned into *2P* blocks. The block Jacobi and HBJ algorithms are based on Hestenes one-sided Jacobi method [16] because it is the best approach for achieving efficient parallel SVD computation (see [12] for more detail).

The rest of this paper is organized as follows. Section 2 evaluates the performance of applying Givens rotation (Level-1 BLAS) on a single node and multiple nodes of Intel Xeon processors. The performance of rank-1 update (Level-2 BLAS) is evaluated in Section 3. Section 4 analyses in details the performance of matrix-matrix multiplication (Level-3 BLAS) on a cluster of Intel Xeon processors. Section 5 discusses in details the performance of SVD based on one-sided Jacobi on a single node and multiple nodes of Intel Xeon processors. The performance of SVD based on HBJ is evaluated in Section 6. Section 7 concludes this paper and gives directions for future work.

2. Performance Evaluation of Applying Givens Rotation (L1 BLAS)

2.1 Superscalar Performance

Applying Givens rotation subroutine is used in many applications including QR decomposition, solving linear system equations, and singular value decomposition (SVD) [17]. In this section, applying Givens rotation is implemented and evaluated on vectors with very small, small, medium, large and very large lengths that ranged (100 – 1000), (1000 – 10,000), (10,000 – 100,000), (100,000 – 1,000,000), and (1,000,000 – 2,000,000), respectively. The content of these vectors are generated randomly to have a value in the interval [1-10]. Figure 2 shows the performances of the superscalar, SIMD, multi-threading, and multi-threading SIMD implementations of applying Givens rotation in GFLOPS on out-of-order superscalar (Intel Xeon E5410) processor. For very small, small, and medium vector lengths fitted in L2 cache, the performance increases as increasing the vector lengths. It shows the superscalar performance increases from 1.75 GFLOPS to 1.96 GFLOPS when the length of the given vector increases from 100 to 1000 (see Figure 2a). Moreover, it is continuing to increase reaching 1.98 GFLOPS on vector length of 10,000 elements (see Figure 2b). For medium vector lengths (10,000 – 100,000), the performance increases until it reaches to 1.99 GFLOPS, as shown in Figure 2c. For large vector lengths



(a) Very small vector lengths (b) Small vector lengths (c) Medium vector lengths

(d) Large vector lengths (e) Very large vector lengths

Figure 2: Performance of applying Givens rotation routine on a single node.

(100,000 – 1,000,000), the performance increases from 1.99 to 2.03 GFLOPS, as shown in Figure 2d. For very large vector lengths (1,000,000 – 2,000,000), the performance still increases until it reaches to 2.1 GFLOPS on vector length 1,500,000. However, when the vector length increases over 1,500,000, the performance decreases to 1.09 GFLOPS (see Figure 2e) because the vectors are not fitted in L2 cache (12 MB). L2 can hold up to 1.5 million single-precision elements for two vectors. Therefore, the rate of cache miss increases and the performance degrades when the vector lengths increase over 1,500,000 elements.

2.2 SIMD Performance

On very small, small, and medium vector lengths fitted in L2 cache, the use of SIMD technique improves the performance of the apply Givens rotation from 2.59 to 5.33 (see Figure 2a), from 5.33 to 6.89 (see Figure 2b), and from 6.89 to 7.36 GFLOPS (see Figure 2c), respectively. It achieves speedups from 1.5 to 2.7, from 2.7 to 3.47, and from 3.47 to 3.7, respectively. On large vector lengths that are still fitted in the L2 cache, the performance improves to reach 7.81 GFLOPS on 1,000,000 elements as shown in Figure 2d, which results in a speedup of 3.84. Increasing the vector length, furthermore, does not improve the performance drastically. A performance of about 8.08 GFLOPS is achieved on 1,500,000 elements, which results in a speedup of about 3.84 representing 96% from the ideal value. However, increasing vector length over 1,500,000 elements, which does not fit in L2 cache, degrades the performance to 3.63 GFLOPS, as shown in Figure 2e. Therefore, the speedup decreases to 3.58 on vector length of 2,000,000 elements.

2.3 Multi-threading SIMD Performance

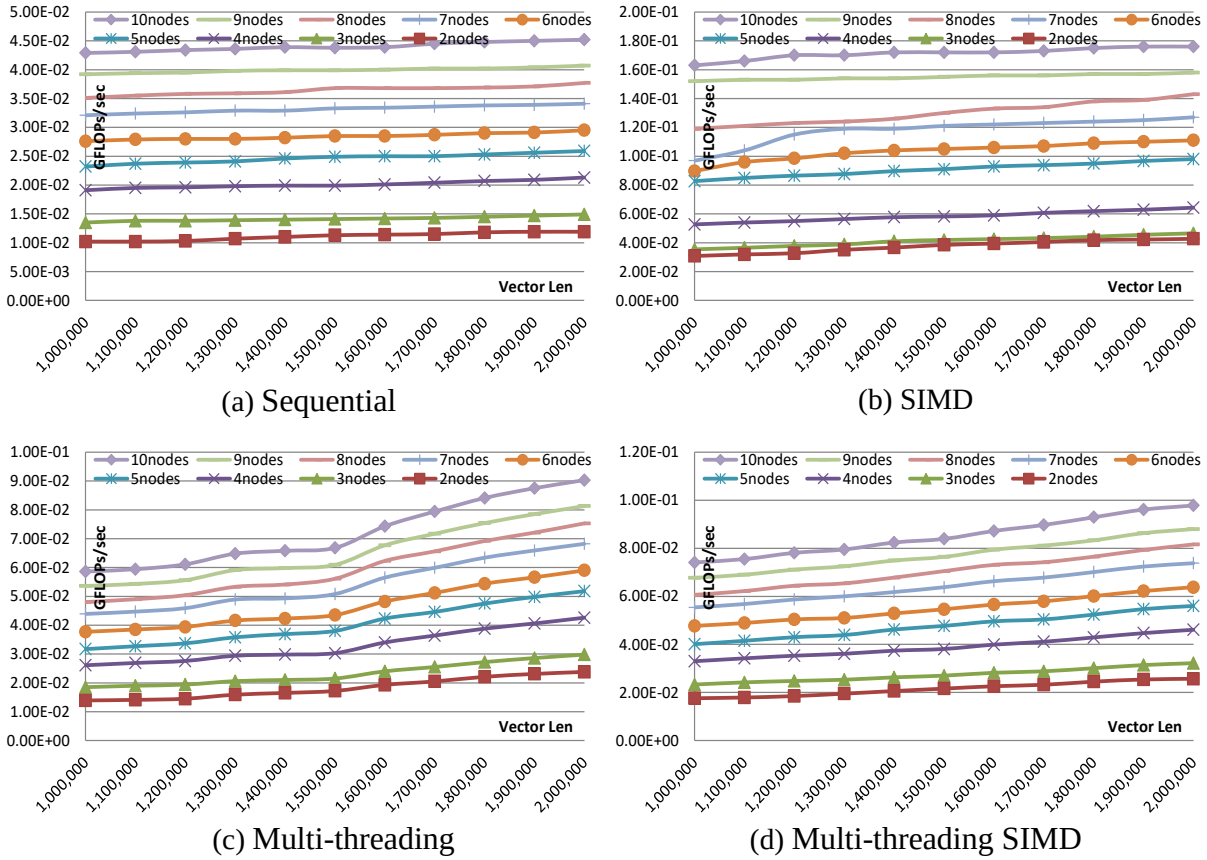
Multi-threading technique does not improve the performance on very small, small, and medium vector lengths because of the thread creation overhead. The performance reaches to 0.018, 0.168, and 1.31 GFLOPS on 1000, 10,000, and 100,000 vector lengths, respectively (see Thrd curve in Figure 2). The performance of multi-threading SIMD implementation (Thrd_SIMD curve in Figure 2) reaches to 0.019, 0.172, and 1.48 GFLOPS at 1000, 10,000, and 100,000 vector lengths, respectively. Note that these performances are worse compared to the sequential (unparallel superscalar) implementation (Seq curve in Figure 2). When the vector lengths are large enough to overcome the threads creation overhead, the performance using multi-threading and multi-threading SIMD techniques improves, as shown in Figures 2d and 2e for large and very large vector lengths, respectively. The performance of the multi-threading implementation increases to 2.58 GFLOPS (for vector length of 1,000,000), and 3 GFLOPS (for vector length of 1,500,000). Such performance contributes to gain speedups of 1.27 and 1.42 for large and very large vector lengths, respectively. For multi-threading SIMD implementation, the performance improves to 3.45 GFLOPS (speedup of 1.69) and to 3.64 GFLOPS (speedup of 1.73) for vector lengths of 1,000,000 and 1,500,000, respectively. When the vector lengths are very large and cannot fit in the L2 cache, the performance begins to decrease because of

increasing cache misses. Applying multi-threading technique on 2,000,000-element vector provides performance of 1.33 GFLOPS, while, multi-threading SIMD technique contributes to a performance of 1.67 GFLOPS for same vector length, as shown in Figure 2e. As the cache miss rate affects the sequential implementation for the very large vector lengths, the speedup is not drastically decreased. For a vector length of 2,000,000, speedup decreases to 1.31 and to 1.65 when applying multi-threading and multi-threading SIMD techniques, respectively. Due to the L2 cache sharing between the four threads, there is a contention on the cache. Therefore the performance of the Thrd_SIMD is lower compared to the SIMD implementation, as shown in Figure 2.

2.4 MPI Performance

The MPI implementations of applying Givens rotation are evaluated on a number of computers ranged from 2 to 10. As each computer does little amount of work ($O(n)$ on n-element vectors), the processing time is too low compared with the sending/receiving time. For example, on very large vector length (2,000,000 elements), the time for sending/receiving data on two nodes is about one second, however, the processing time is 2.15×10^{-3}, 6.89×10^{-4}, 9.42×10^{-4}, and 8.89×10^{-4} seconds for the sequential, SIMD, multi-threading, and multi-threading SIMD implementations, respectively. Thus, the performance of applying Givens rotation decades due to using MPI on distributed system.

Figure 3a shows the performance in GFLOPS of the sequential implementation on 2 to 10 computers. As shown in Figure 2, the maximum performance on one node on vector



(a) Sequential (b) SIMD

(c) Multi-threading (d) Multi-threading SIMD

Figure 3: Performance of applying Givens rotation on multiple nodes.

length of 1,500,000 elements is 2.1 GFLOPS. This performance decreases to 1.13×10^{-2}, 1.41×10^{-2}, 1.99×10^{-2}, 2.49×10^{-2}, 2.85×10^{-2}, 3.33×10^{-2}, 3.68×10^{-2}, 3.99×10^{-2}, and 4.38×10^{-2} GFLOPS on 2, 3, 4, 5, 6, 7, 8, 9, and 10 nodes, respectively. The performance of the SIMD implementation on multiple nodes of computers decreases from 8.08 GFLOPS to 3.85×10^{-2}, 4.19×10^{-2}, 5.82×10^{-2}, 9.10×10^{-2}, 1.05×10^{-1}, 1.21×10^{-1}, 1.30×10^{-1}, 1.55×10^{-1}, and 1.72×10^{-1} GFLOPS, respectively, see Figure 3b. The performance of the multi-threading implementation on 2 to 10 computers decreases to 1.72×10^{-2}, 2.15×10^{-2}, 3.03×10^{-2}, 3.80×10^{-2}, 4.35×10^{-2}, 5.08×10^{-2}, 5.61×10^{-2}, 6.08×10^{-2}, and 6.68×10^{-2} GFLOPS, respectively, see Figure 3c, where the maximum performance on the same vector length (1,5000,000 elements) on a single node is 3 GFLOPS. Finally, the performance of the multi-threading SIMD implementation decreases from 3.64 GFLOPS to 2.16×10^{-2}, 2.70×10^{-2}, 3.81×10^{-2}, 4.77×10^{-2}, 5.46×10^{-2}, 6.38×10^{-2}, 7.05×10^{-2}, 7.64×10^{-2}, and 8.39×10^{-2}, respectively, as shown in Figure 3d.

3. Performance Evaluation of Rank-1 Update (L2 BLAS)

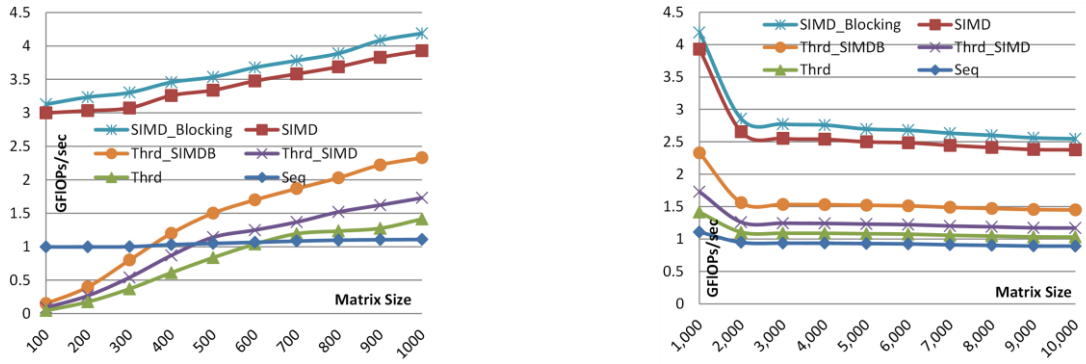
3.1 Superscalar Performance

For small matrix sizes, the performance of rank-1 update enhances with increasing the matrix size. The performance of the *ij* variant improves until 1.11 GFLOPS at matrix size 1000×1000 (see Figure 4a). However, the performance decreases for large matrix sizes because of the cache miss, see Figure 4b.

3.2 SIMD Performance

Due to using SIMD technique, the performance of the rank-1 update is improved with increasing matrix sizes fitted in the L2 cache. On small matrix size (1000×1000), the performance of rank-1 update increases reaching 3.93 GFLOPS (see SIMD curve in Figure 4a). This results in a speedup of 3.54. However, on large matrix sizes, that are too large to fit in the L2 cache (larger than 2000×2000), the performance of rank-1 update degrades because of the escalating cache miss rate, as shown in Figure 4b.

The performance can be improved, furthermore, using the blocking technique, which provides the ability of reusing the loaded vector date many times. It leads to decreasing the number of memory references. For rank-1 update based on SAXPY (scalar s times vector x plus vector y), the number of memory references decreases from $(3n^2 + n)$ (the worst case when using the loaded vector data only once) to $(2n^2 + 2n)$ (the best case when reusing the loaded vector data many times, but it is not practical for large vector lengths). Thus, the number of memory references depends on the number of the temporary registers that are used to reusing the loaded data. A better performance is achievable with larger temporary registers number used, because of the higher level of reusing the loaded vector data.



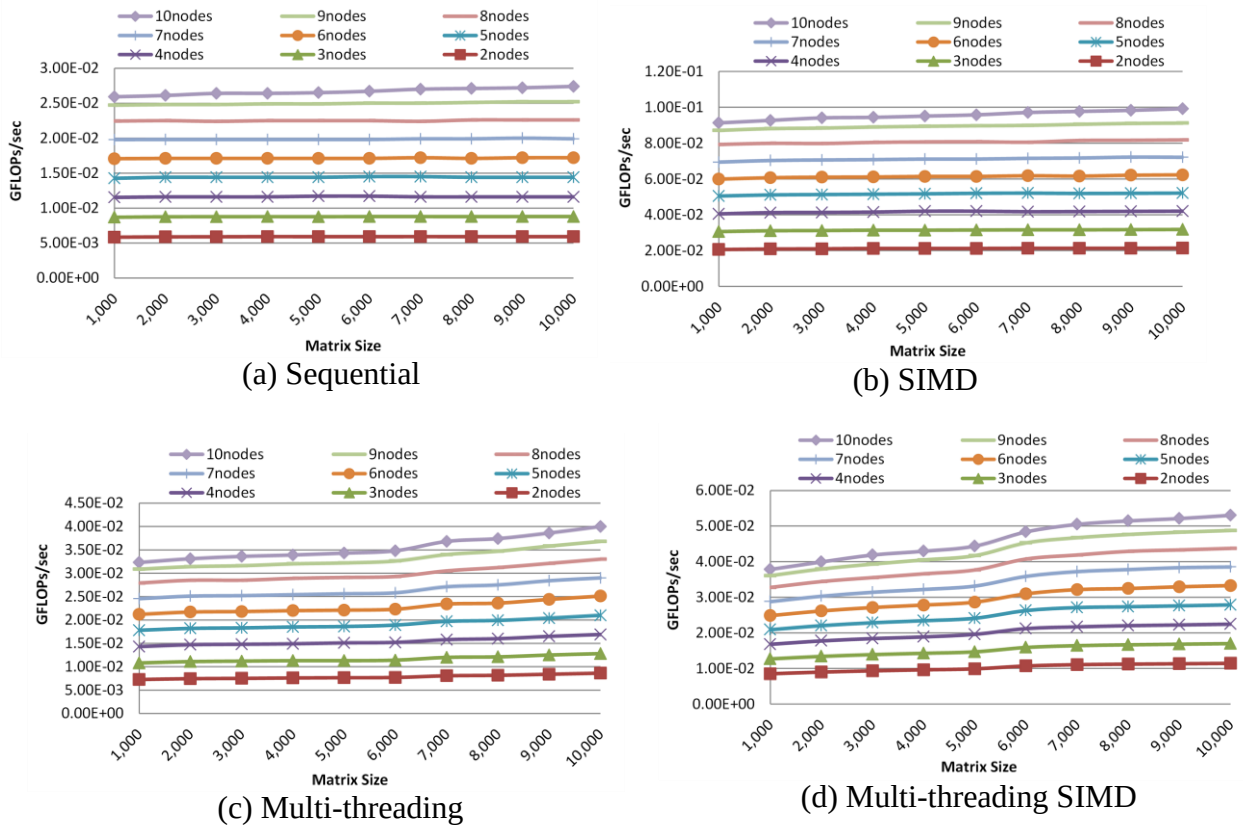
(a) Small matrix sizes (b) Large matrix sizes

Figure 4: Performance of rank-1 update subroutine on a single node.

Attributable to using the blocking technique, the performance of the rank-1 update is enhanced even more. On matrix size of 1000×1000 elements, its performance increases to 4.19 GFLOPS, as shown in Figure 4a (SIMD_Blocking curve). Moreover, the speedup improves to 3.78. As the performance depends on the size of the cache memory, the performance decreases at large matrix sizes because of growing penalty of cache miss and the loaded matrix is used only once, see Figure 4b (SIMD_Blocking curve). In conclusion, the performance depends on the size of the cache memory and the number of the temporary SIMD registers, using the higher ones gives better performance.

3.3 Multi-threading SIMD Performance

Figure 4 shows the improvements in the performance when using multi-threading (Thrd), multi-threading SIMD (Thrd_SIMD), and multi-threading SIMD blocking (Thrd_SIMDB) techniques. As expected, the performance speeds down for very small matrix sizes because of the thread creation overhead. With increasing the matrix size, the performance improves, where the effect of the thread creation overhead is decreased. The performance keeps the growing trend until matrix size reaches 1000×1000 elements, which is the maximum size that can fit in the L2 cache. The performance of the multi-threading, multi-threading SIMD, and multi-threading SIMD blocking reaches 1.41, 1.73, and 2.33 GFLOPS, respectively (Figure 4a). The speedups increase to 1.27, 1.56, and 2.10, respectively. Because the performance of Level-2 BLAS depends on the size of the cache memory, it decreases at large matrix size, which is too large to fit in the L2 cache, (for all the implementations) due to increased cache miss rate, as shown in Figure 4b.



(a) Sequential (b) SIMD

(c) Multi-threading (d) Multi-threading SIMD

Figure 5: Performance evaluation of rank-1 update on 2 to 10 nodes.

3.4 MPI Performance

As the amount of transmitted/received data is larger than the number of floating-point operations that are computed on each node, the sending/receiving time dominates the overall time. The processing times of sequential, SIMD, multi-threading, and multi-threading SIMD implementations on 2 nodes for rank-1 update are 1.01×10^{-1}, 2.71×10^{-2}, 5.78×10^{-2}, and 4.36×10^{-2} seconds, respectively.

As in Level-1, Level-2 BLAS do not gain a speedup in performance on multiple computers because of the high overhead of sending/receiving time between them. On rank-1 update, the performance speeds down. Its performance on one node for the sequential, SIMD, multi-threading, and multi-threading SIMD (on matrix size of 1000×1000) is 1.11, 3.93, 1.41, and 1.73 GFLOPS, respectively. When executing these implementations on 2/5/10 computers, the performance decreases to 5.83×10^{-3}/1.43×10^{-2}/2.59×10^{-2} for the sequential implementation, as shown in Figure 5a. For the SIMD implementation, the performance drops to 2.06×10^{-2}/5.05×10^{-2}/9.13×10^{-2} GFLOPS, respectively, as shown in Figure 5b. Moreover, the performance of the multi-threading implementation degrades to 7.26×10^{-3}/1.78×10^{-2}/3.23×10^{-2} GFLOPS, respectively, as shown in Figure 5c. Finally, the multi-threading SIMD implementation performance drops to 8.51×10^{-3}/2.09×10^{-2}/3.78×10^{-2} GFLOPS, as shown in Figure 5d.

Although, the performance decreases with applying MPI technique on Level-2 BLAS, the performance of the SIMD implementation on multiple nodes is higher than the multi-threading and multi-threading SIMD implementations. This is because of the threads

creation overhead and the fighting for the cache between threads. For example the performances of the SIMD, multi-threading, and multi-threading SIMD for the rank-1 update subroutine executed on 10 nodes on large matrix size of 10,000×10,000 are 9.92×10^{-2}, 4×10^{-2}, and 5.3×10^{-2} GFLOPS, respectively, see Figure 5.

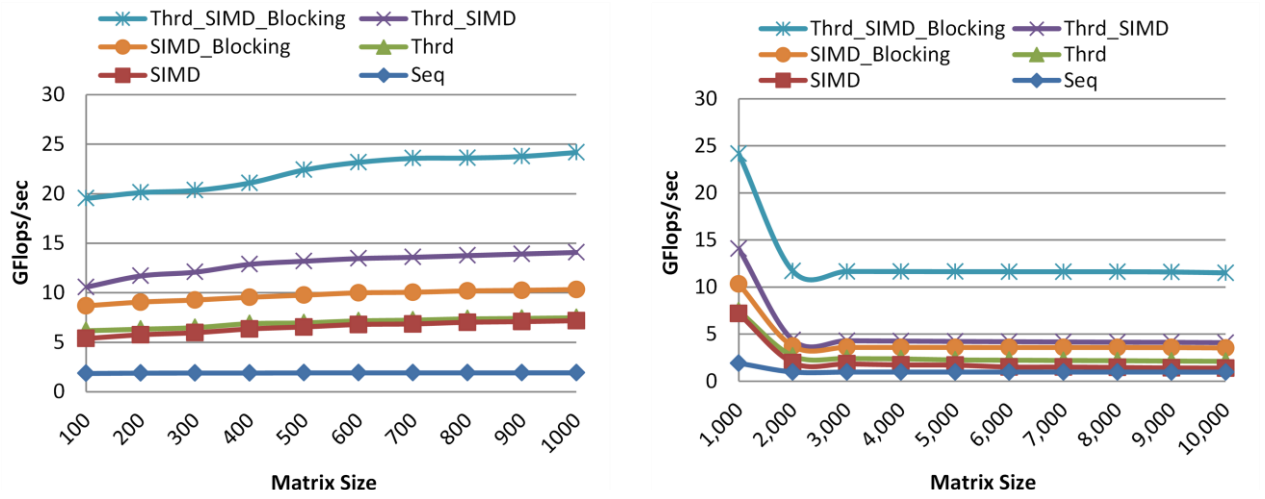
4. Performance Evaluation of Matrix-Matrix Multiplication (L3 BLAS)

4.1 Superscalar Performance

On small matrix sizes (100×100 – 1000×1000), the performance of the best variant (*ikj*) of matrix-matrix multiplication reaches to 1.91 GFLOPS at matrix size 1000×1000, as shown in Figure 6a. However, when the matrix sizes became larger than 1000×1000, the performance degrades, because larger sizes cannot fit in the L2 cache. The rate of cache misses increases and the superscalar performance of the traditional matrix-matrix multiplication degrades with increasing size of matrices over 1000×1000.

4.2 SIMD Performance

To take advantage of the DLP existing in the traditional algorithm of matrix-matrix multiplication, SIMD technique is applied (see Figure 6) for parallel processing four single-precision (32-bits) elements using a single instruction. The *ikj* variant that is based on SXAPY permits the efficient use of SIMD technique as four elements in the same row from matrix *B* are loaded and multiplied by a single element of the matrix *A*, simultaneously. As a result of using the SIMD technique, the performance of the traditional algorithm of the matrix-matrix multiplication improves as follows. On small matrix size (100×100), the performance is 5.38 GFLOPS, while it reaches to 1.84 GFLOPS for the sequential superscalar implementation. As the matrix size increases the performance improves to 7.18 GFLOPS on 1000×1000. It shows that applying the SIMD technique achieves a maximum speedup of 3.76 at the maximum size that can fit in the L2 cache. This speedup represents 94% of the ideal performance. Because of the growing cache miss rate for large matrix sizes, which cannot fit in the L2 cache, the SIMD performance of matrix-matrix multiplication begins to decrease, as shown in Figure 6b.



(a) Small matrix sizes

(b) Large matrix sizes

Figure 6: Performance of matrix-matrix multiplication on a single node.

Beside SIMD technique, exploiting the memory hierarchy enhances the performance even more. Figure 6 (SIMD_Blocking curve) shows the improvement in the performance due to exploiting the memory hierarchy, where the performance on the small matrix sizes increases to 8.68 GFLOPS on 100×100 and 10.33 GFLOPS at 1000×1000 (see Figure 6a SIMD_Blocking curve). This corresponds to a maximum speedup of 5.41 at 1000×1000. Although, the performance decreases for large matrix sizes, it is better compared to the case using SIMD technique, alone, for the same sizes. This is because of reusing the loaded data in the cache offered by the matrix blocking technique leading to reducing the cache miss rate. An average speedup of 3.71 is achieved for large matrix sizes, which is considered 92.75% of the ideal speedup of the SIMD technique.

4.3 Multi-threading SIMD Performance

On small matrix sizes, the multi-threading performance improves with increasing the matrix size until it reaches 7.48 GFLOPS on matrix size of 1000×1000 elements, achieving a speedup of 3.91, which represents 97.75% of the ideal value, see Figure 6a (Thrd curve). Moreover, each thread can compute its operations in parallel using SIMD instructions. Figure 6a (Thrd_SIMD curve) shows how the combination of the two techniques improves the performance. The performance increases to 14.07 GFLOPS on matrix size 1000×1000. It results in a speedup of 7.37, (46.1% from the ideal value), where the ideal speedup is 16 in the case of combining the multi-threading and SIMD techniques.

Since the cache memory is shared between the parallel threads, the performance of the multi-threading and multi-threading SIMD implementations degrades on large matrix sizes because of increasing the rate of cache misses. Figure 6b (Thrd and Thrd_SIMD curves) shows the degradation in performance for large matrices that cannot be fitted in the L2 cache. To improve their performance through reducing the rate of cache miss, each thread exploits the memory hierarchy using the matrix blocking technique (see Figure 6).

As explained earlier, matrix blocking technique reduces the load/store operations from/to the main memory and reuses the loaded data in the cache memory many times. Therefore, the combination between the multi-threading, SIMD, and matrix blocking techniques leads to a huge improvement in the performance of the traditional algorithm of matrix-matrix multiplication. It reaches to 24.17 GFLOPS on matrix size of 1000×1000 achieving a speedup of 12.65 (79.1 % of the ideal) over the single threaded sequential implementation. Moreover, for large matrix size, the performance is enhanced, leading to an average performance of 11.63 GFLOPS and an average speedup of 12.02 (75.13% of the ideal), as shown in Figure 6 (Thrd_SIMD_Blocking curve).

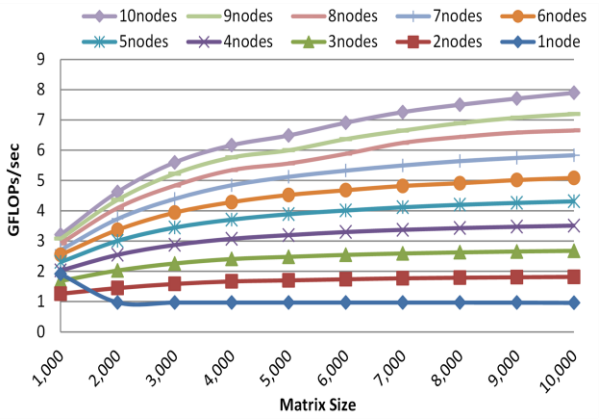
4.4 MPI Performance

All forms of parallelism (ILP, DLP, and TLP) can be applied on multiple nodes to achieve improved performance. Multi-threading, SIMD, and matrix blocking techniques are used on each node. In contrast to Level-1 and Level-2 BLAS, the total time of broadcasting, sending, and receiving data is negligible compared to the processing time of Level-3 BLAS on large matrix sizes.

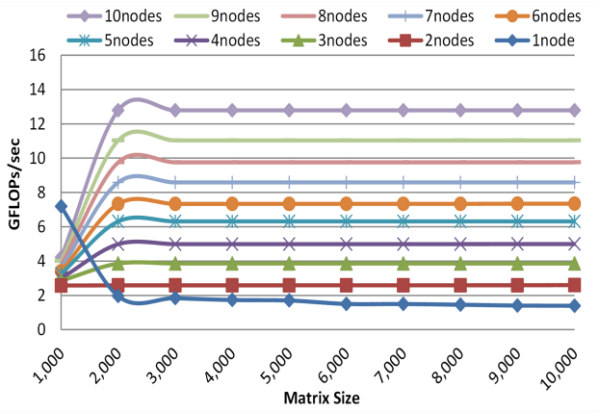
On small matrix sizes, each node takes a small part from the input matrix *A* to multiply by matrix *B*. For example, if the size of the input matrices (*A* and *B*) is 100×100 and the number of nodes is 10, each node receives the whole matrix *B* (100×100) and a sub-matrix (10×100 elements) of matrix *A*. However, on large matrix sizes (like 10,000×10,000); the processing time is very large compared with the time spent in sending/receiving data through the network. The arithmetic operations done on each node consume time large enough to hide the overhead time of the network for all implementations. For example in the sequential (Seq) implementation, the processing times are 1033, 689, 516, 413, 344, 295, 227, 219, and 206 seconds on 2, 3, 4, 5, 6, 7, 8, 9, and 10 nodes, respectively. Furthermore, applying all parallel processing techniques together decreases the processing time drastically. On 10 nodes, the processing times are reduced to 206, 119, 75, 38, and 6 seconds for the sequential, SIMD, multi-threading, multi-threading SIMD, and multi-threading SIMD blocking implementations, respectively.

Based on the above discussion, the use of MPI technique on small matrix sizes speeds down the performance of the traditional matrix-matrix multiplication algorithm. However, on large matrix sizes, a good performance is achievable due to using MPI and sharing the execution between more than one computer. Figure 7 shows the performances in GFLOPS of the sequential, SIMD, multi-threading, multi-threading SIMD, and multi-threading SIMD blocking implementations, respectively on large matrix sizes.

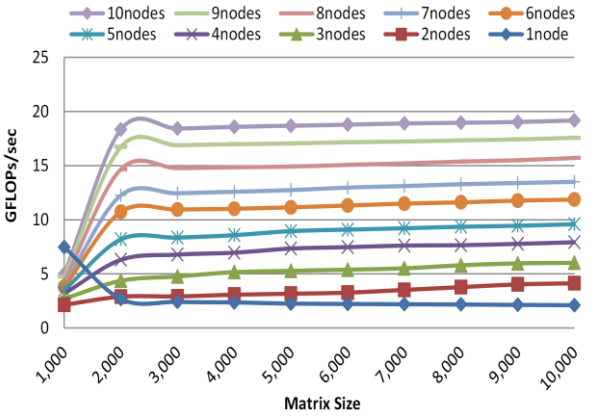
Firstly, for the sequential implementation of matrix size of 10,000×10,000, the performance increases from 0.96 GFLOPS on one node to 1.82, 2.67, 3.51, 4.31, 5.08, 5.83, 6.65, 7.19, and 7.89 GFLOPS on 2, 3, 4, 5, 6, 7, 8, 9, and 10 nodes, respectively, see Figure 7a. These improvements provide speedups of 1.89, 2.79, 3.66, 4.49, 5.30, 6.08, 6.94, 7.51, and 8.24, respectively over the sequential implementation on one node. These



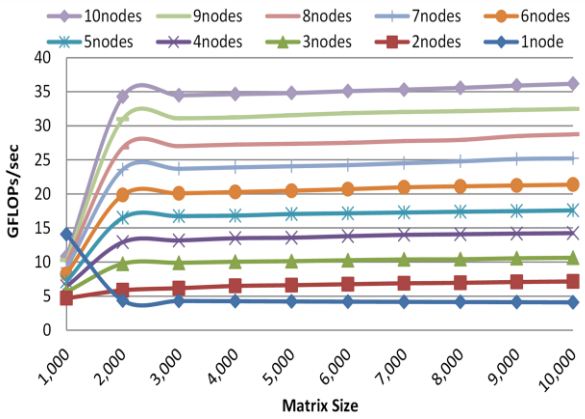
(a) Sequential



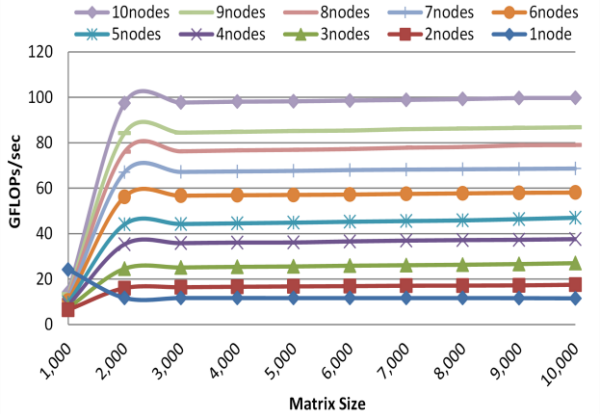
(b) SIMD



(c) Multi-threading



(d) Multi-threading SIMD



(e) Multi-threading SIMD blocking

Figure 7: Performance evaluation of matrix multiplication algorithm on 2 to 10 nodes.

speedups represent 94.5%, 93%, 91.5%, 89.8%, 88.3%, 86.8%, 86.7%, 83.4%, and 82.4%, respectively, of the ideal speedup. Note that as increasing the number of nodes, the speedup is far away from the ideal value because more network overhead is added when increasing the number of nodes.

Secondly, for the SIMD implementation, Figure 7b shows how the performance improves, furthermore, when applying the SIMD technique in addition to the MPI. On matrix size of 10,000×10,000, the performance reaches to 2.59, 3.86, 4.99, 6.31, 7.35, 8.59, 9.75, 11.03, and 12.79 GFLOPS, respectively, see Figure 7b. However, on one node, the performance of the SIMD implementation was 1.39 GFLOPS. Using the SIMD technique in addition to MPI achieve speedups of 1.86, 2.77, 3.58, 4.53, 5.27, 6.15, 6.99, 7.91, and 9.17, respectively, over the SIMD implementation on one node. The speedups over the sequential implementation on one node improve to 2.71, 4.02, 5.21, 6.59, 7.66, 8.96, 10.18, 11.51, and 13.34, respectively on matrix size 10,000×10,000.

The utilization of the multi-threading technique in addition to the MPI technique can improve the performance furthermore. Figure 7c shows the performances in GFLOPS of the multi-threading implementation on 2 to 10 nodes on large matrix sizes. On matrix size of 10,000×10,000, the performance of the multi-threading on one node improves from 2.11 GFLOPS to 4.13, 6.02, 7.92, 9.59, 11.86, 13.49, 15.70, 17.56, and 19.17 GFLOPS, respectively. Thus, the speedups gained using the multi-threading technique on many computers over the performance on a single computer, using same technique, are 1.96, 2.86, 3.76, 4.55, 5.63, 6.41, 7.45, 8.33, and 9.09, respectively. In addition, speedups of 4.31, 6.28, 8.26, 10, 12.37, 14.08, 16.38, 18.32, and 20 are achieved over the sequential implementation on one node on matrix size 10,000×10,000.

Although multi-threading or SIMD technique improves the performance of the traditional algorithm of the matrix multiplication, the combination between these two techniques can accelerate the execution drastically. In this case, the single node with multiple threads can adopt SIMD instructions inside each one of them. Employing such combination on matrix size of 10,000×10,000 leads to performance improves to 7.15 (2 nodes), 10.63 (3 nodes), 14.25 (4 nodes), 17.59 (5 nodes), 21.36 (6 nodes), 25.22 (7 nodes), 28.74 (8 nodes), 32.48 (9 nodes), and 36.16 (10 nodes) GFLOPS compared to 4.09 GFLOPS on a single node, see Figure 7d. Based on this, speedups of 1.75, 2.59, 3.48, 4.29, 5.22, 6.16, 7.02, 7.94, and 8.84, respectively, are achieved over the multi-threading SIMD implementation on a single node. Moreover, speedups over the sequential implementation on the single node are 7.46, 11.09, 14.86, 18.35, 22.29, 26.30, 29.99, 33.88, and 37.73, respectively.

Finally, in addition to MPI, multi-threading, and SIMD techniques, the matrix blocking can be used to exploit the memory hierarchy on each node and reuse the loaded data in the cache many times. Exploiting all forms of parallelism gives the best performance, where the performances reach to maximum values for matrix size of 10,000×10,000, which are as follow: 17.49, 26.99, 37.58, 46.99, 58.09, 68.67, 78.99, 86.80, and 99.73 GFLOPS, respectively, as shown in Figure 7e, while the performance on single node is 11.51 GFLOPS. The speedups compared to the multi-threading SIMD blocking implementation on single node are 1.52, 2.35, 3.27, 4.08, 5.05, 5.97, 6.86, 7.54, and 8.67, respectively. Due to this large improvement in the performances, the speedups over the

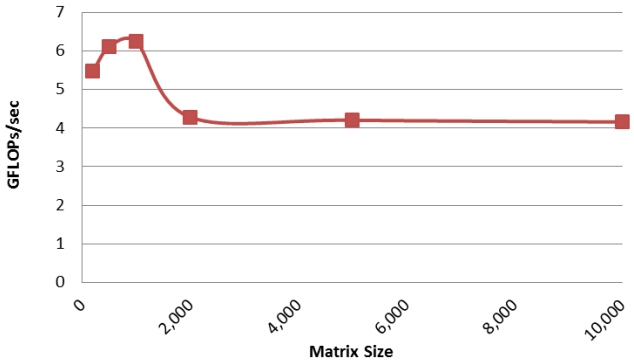


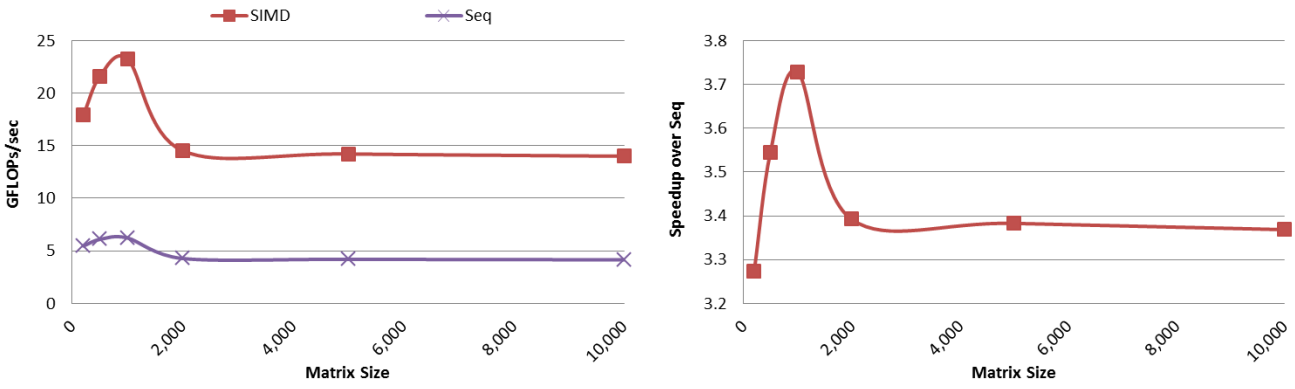
Figure 8: Superscalar performance in GFLOPS of the unparallel one-sided Jacobi algorithm.

sequential implementation on one node increase to 18.25, 28.16, 39.20, 49.03, 60.59, 71.63, 82.41, 90.55, and 104.04, respectively.

5. Performance Evaluation of SVD based on One-Sided Jacobi

5.1 Superscalar Performance

The one-sided Jacobi algorithm is implemented and evaluated on matrices with small matrix sizes (200×200, 500×500, and 1000×1000) that can fit in the L2 cache, and the large matrix sizes (2000×2000, 5000×5000, and 10,000×10,000) that cannot fit in the L2 cache. These matrices hold single-precision floating-point (SP-FLOP) numbers, and their content is generated randomly to have a value in the interval [1, 10]. Besides, the norm of the input matrix times 10^{-7} is used as the convergence condition for single-precision data. Figure 8 shows the superscalar performance of the sequential implementation of one-sided Jacobi algorithm for SP-FLOP numbers. An average performance of 5.93 GFLOPS is achieved on the target processor through the only exploitation of ILP using superscalar execution, for the small matrix sizes. However, for



(a) Performance in GFLOPS (b) Speedup over OSJ Sequential implementation

Figure 9: SIMD performance of one-sided Jacobi algorithm.

the large matrix sizes, the average performance decreases to 4.21 GFLOPS because these matrices are too large to fit in the L2 cache, and so that the rate of cache misses increases (3 million single-precision data can hold in L2 cache).

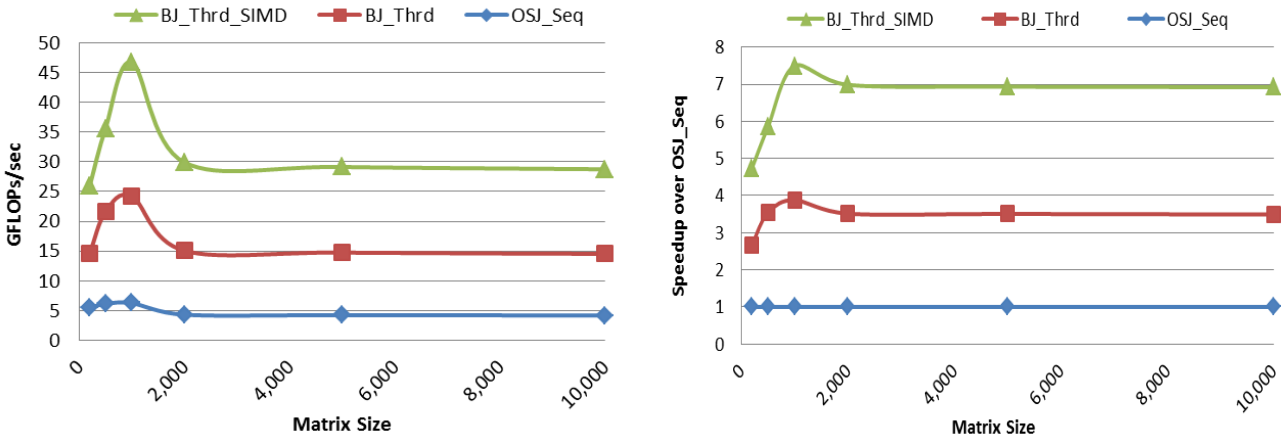
5.2 SIMD Performance

Figure 9 shows the SIMD performance of one-sided Jacobi algorithm. Applying SIMD technique improves the performance of the sequential superscalar implementation, and the performance increases with increasing the matrix size of single-precision data. At small matrix sizes, the average performance improves to 20.93 GFLOPS (see SIMD curve) that achieve an average speedup of 3.52 as shown in Figure 9b, which is considered 88% of the ideal speedup. Because of the high cache miss rate at large matrix sizes, which cannot fit in the L2 cache, the performance begins to decrease for the single-precision data, but it is still better than the sequential implementation due to using the SIMD technique.

5.3 Multi-threading SIMD Performance

Figure 10 (BJ_Thrd curve) shows the performance of the multi-threading block Jacobi algorithm on quad-core Intel Xeon E5410 processor, where the average performance improves to 20.15 GFLOPS. It leads to an average speedup of 3.36 over the OSJ sequential implementation (around 84% from the ideal) at small matrix sizes. Because of the cache misses overhead at large matrix sizes, which are so large to fit in the L2 cache, the performance degrades, where the average performance decreases to 14.77 GFLOPS.

Besides using multi-threading technique, to improve the performance of the block Jacobi, DLP can be exploited to achieve more improvement using SIMD instructions. In our implementation, each thread executes its task (calculating and applying the rotation parameters (c, s)) using SIMD technology. Figure 10 (BJ_Thrd_SIMD curve) shows the performance of the multi-threading SIMD implementation. For single-precision data, processing four elements in the same time using single SIMD instruction enhances the performance for small matrix sizes to an average of about 36.11 GFLOPS. It achieves an average speedup of 6.03 over single threaded OSJ, that represents 37.69% from the ideal



(a) Performance in GFLOPS

(b) Speedup over OSJ_Seq

Figure 10: Performance of the multi-threading block Jacobi algorithm.

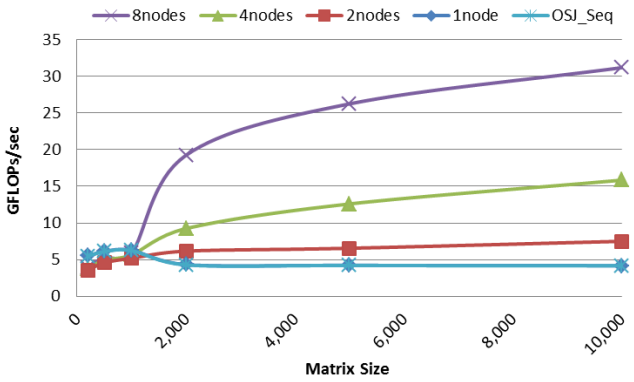
value (ideal speedup is 16 due to using multi-threading SIMD for the single-precision data). Moreover, for large matrix sizes, not only increasing the rate of cache misses affects the performance of the multi-threading SIMD implementation, but also the time that each thread has to wait for working on the 128-bit registers as the target processor has only sixteen 128-bit registers for SIMD operations while there is four threads run in parallel. These reasons cause the performance to degrade for such large sizes.

5.4 MPI Performance

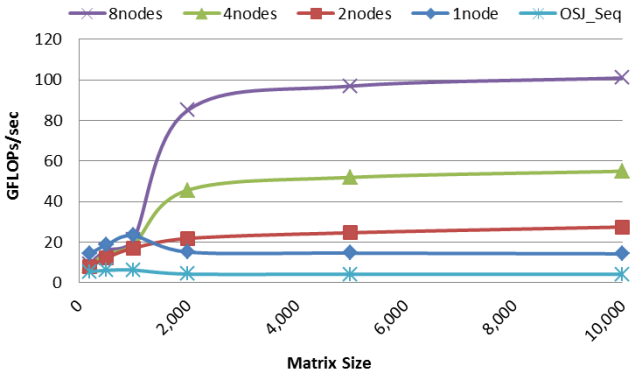
Processing matrices with large sizes can diminish the overhead of the network and can gain better performance on multiple nodes. When using one computer, the performance of the sequential implementation of the block Jacobi is close to the performance of the OSJ algorithm. That is because when the whole matrix is processed by a single computer, the better performance is gained at the maximum size that can fit in the cache, and then the performance degrades because of the cache miss penalty. When using a single computer, the sequential implementation's performance is 6.29 GFLOPS on matrix size 1000×1000, and when the matrix size increases to 10,000×10,000, the performance decreases to 4.18 GFLOPS for the single-precision data. On large matrix sizes (10,000×10,000), using 2, 4, 8 computers lead to improvements in the performance of the sequential implementation of the block Jacobi algorithm to 7.49, 15.84, and 31.18 GFLOPS, respectively, see Figure 11a. These performances achieve speedups of 1.80, 3.81, and 7.50, respectively, over the performance of the sequential implementation of the OSJ algorithm. These speedups are considered to be 90%, 95.25%, and 93.75% of the ideal value.

Due to using SIMD, the performance on single computer of matrix size of 1000×1000 increases to 23.24 GFLOPS, which achieve speedup of 3.73 over the sequential implementation of OSJ, this speedup represents 93.25% of the ideal value. On 2, 4, and 8 computers, applying the SIMD technique in addition to MPI enhances the performance of the block Jacobi algorithm on large matrix size of 10,000×10,000 to 27.47, 54.96 and 100.95 GFLOPS, respectively, see Figure 11b. Thus, speedups of 6.61, 13.22, and 24.28, respectively, are achieved over the performance of the OSJ algorithm.

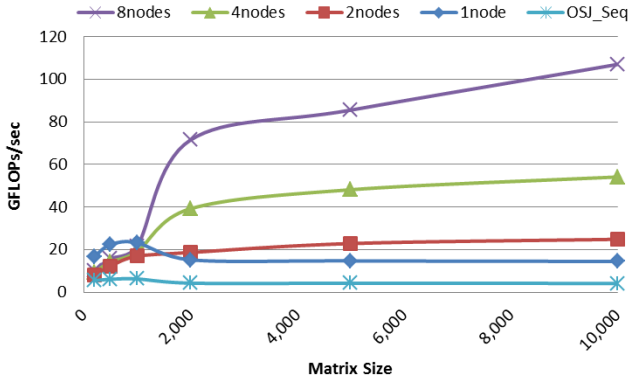
Figure 11c shows how the performance of the block Jacobi algorithm is improved using multi-threading technique on each multi-core processor in the cluster. Because of the cache miss, the performance on one node reaches 23.25 GFLOPS, which is the maximum performance that can be achieved on matrix size of 1000×1000. 1000×1000 is the largest matrix size can fit in the L2 cache. Increasing the matrix size furthermore results in degrading the performance. However, the performance improves on large matrix sizes on 2, 4, and 8 computers, each applies the multi-threading technique. On matrix size of 10,000×10,000, the performance increases to 24.86, 54.12, and 106.97 GFLOPS, respectively. As a result of this improvement, the speedups over the OSJ algorithm enhance to 5.98, 13.02, and 25.73, respectively.



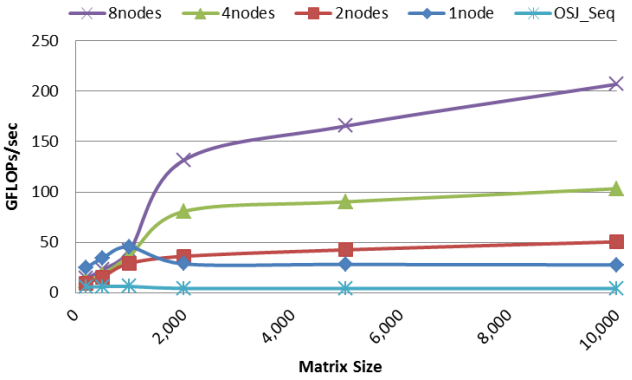
(a) Sequential performance



(b) SIMD performance



(c) Multi-threading performance



(d) Multi-threading SIMD performance

Figure 11: Performance of the parallel block Jacobi algorithm on a cluster of 2, 4, and 8 nodes.

Moreover, using the SIMD technique in addition to the multi-threading technique on each computer of the cluster enhances the performance even more, because of three forms of parallel processing techniques being combined together. Using this combination improves the performance on one node to 45.55 GFLOPS on matrix size 1000×1000, which achieve speedup of 7.30 (45.6% of the ideal value). When the matrix size increases more (cannot fit in the L2 cache), the performance on one node decreases as increasing the rate of cache miss. The performance degrades to 27.67 GFLOPS on large matrix size of 10,000×10,000.

Partitioning the input matrix into blocks and distributed these blocks among the computers omits the effect of cache miss on large matrix sizes. Thus, employing multi-threading and SIMD techniques on 2, 4, and 8 computers accelerates the execution of the block Jacobi algorithm on large matrix sizes. On matrix size of 10,000×10,000, the performance increases to 50.49, 103.26, and 206.97 GFLOPS, as shown in Figure 11d. Therefore, the speedup over the OSJ algorithm enhances to 12.15, 24.84, and 49.79, respectively.

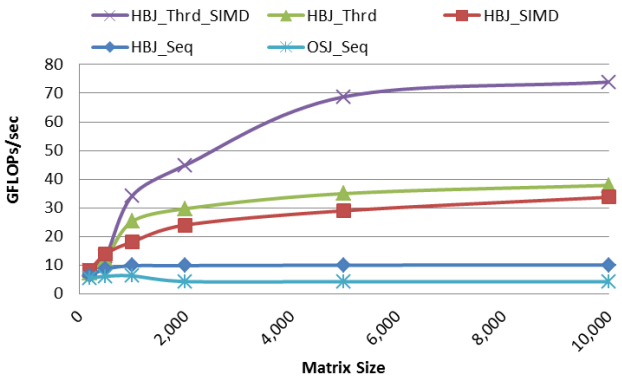
6. Performance Evaluation of SVD based on Hierarchal Block Jacobi

6.1 Superscalar Performance

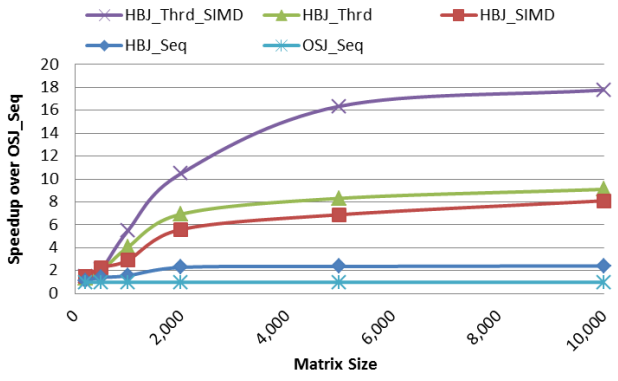
Because of exploiting the memory hierarchy, reusing the data been hold in the cache many times, and processing blocks of data instead of vectors, improve the performance of the OSJ algorithm. Moreover, the performance does not decrease on large matrices because the rate of cache misses decreases. Figure 12 shows the performance of the HBJ algorithm on the Intel Xeon processor for single-precision data. On large matrix size, 10,000×10,000, the performance increases to 10.06 GFLOPS that achieves a speedup of 2.42 over the OSJ sequential implementation (OSJ_Seq) (see Figure 12 HBJ_Seq curves).

6.2 SIMD Performance

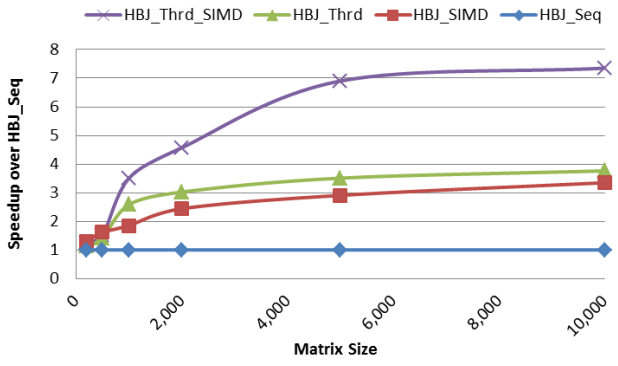
Applying the SIMD technique on the HBJ for processing multiple elements in parallel improves the performance furthermore. The performance increases to 33.69 GFLOPS that achieves speedup of 8.11 over the OSJ_Seq, and 3.35 over the HBJ_Seq. This represents 83.75% of the ideal performance (see Figure 12 HBJ_SIMD curves).



(a) Performance in GFLOPS



(b) Speedup over OSJ_Seq



(c) Speedup over HBJ_Seq

Figure 12: Performance of the multi-threading hierarchal block Jacobi algorithm.

6.3 Multi-threading SIMD Performance

Exploiting TLP on quad-core processor by creating four threads to run in parallel on the four cores, where each thread orthogonalizes the super-rows of two blocks of the parallel HBJ algorithm, enhances the performance of the HBJ. At large matrix size 10,000×10,000, using the multi-threading technique increases the performance to 37.85 GFLOPS. Therefore, speedups of 9.11 over the OSJ_Seq, and 3.76 over HBJ_Seq are achievable. It represents 94% of the ideal performance (see Figure 12 HBJ_Thrd curves). Moreover, the performance can be enhanced more when combining the multi-threading and SIMD techniques, where each thread apply the SIMD technique individually. Thus, the performance in GFLOPS increases to 73.91, which achieve speedups of 17.78 over the OSJ_Seq, and 7.35 over HBJ_Seq (see Figure 12 HBJ_Thrd_SIMD curves).

6.4 MPI Performance

The performance evaluation of MPI implementations of HBJ on a cluster of 2, 4, and 8 computers is presented in Figure 13 for single-precision data. For the sequential implementation of the HBJ algorithm on clusters of 2, 4, and 8 computers, the performance, for large matrix size of 10,000×10,000 increases from 10.11 on one node to 17.71, 36.41, and 76.54 GFLOPS respectively, (see Figure 13a). As a result of this improvement, the speedup over the OSJ increases from 2.43 to 4.26, 8.76, and 18.41 respectively. Measuring the speedups of the performance relative to the sequential implementation on a single computer (HBJ_Seq_1node), on 2, 4, and 8 computers the speedups are 1.75, 3.60, and 7.57, which represent 87.5%, 90%, and 94.6%, respectively, of the ideal performance.

To further improve performance of HBJ on a cluster of Xeon processors, SIMD instructions are applied on HBJ. The performance of our implementation of HBJ algorithm using SIMD in addition to MPI techniques on clusters with 2, 4, and 8 computers improves from 33.79 on one node to 60.19, 118.65, and 259.97 GFLOPS, respectively, on large matrix size of 10,000×10,000, as shown in Figure 13b. Thus, the speedups over the performance of the OSJ_Seq improve from 8.13 to 14.48, 28.54, and 62.54, respectively. While the performance on 2, 4, and 8 computers achieve a speedup of 1.78, 3.51, and 7.69 respectively over the HBJ_SIMD on single computer, which represents 89%, 87.75%, and 96.12% respectively, from the ideal speedup.

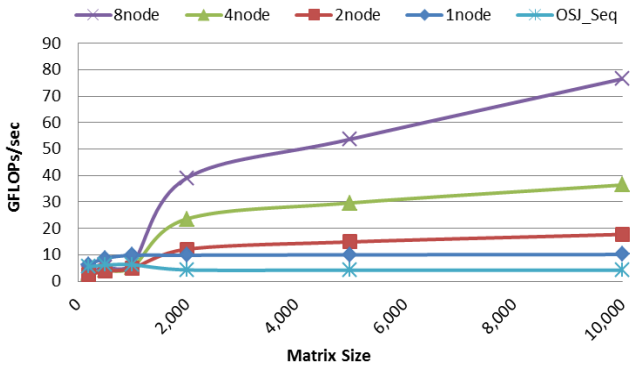
In addition to partition the blocks of the parallel HBJ algorithm between more than one computer, each computer can divide its blocks into sub-blocks and processes them in parallel by creating four threads, each thread run on a core of the Intel Xeon quad-core processor. As a result of this combination between the MPI and multi-threading techniques, a good performance of the parallel HBJ is achieved. On large matrix size of 10,000×10,000, applying multi-threading technique achieves high performance of about 37.98 GFLOPS on one node, and it increases to 67.33, 139.99, and 295.16 single-precision GFLOPS on clusters with 2, 4, and 8 computers, respectively (see Figure 13c). Moreover, the speedups over unparalleled OSJ increase from 9.13 to 16.19, 33.68, and 71, respectively. The

speedups over HBJ_Thrd_1node are 1.77, 3.69, and 7.77 on 2, 4, and 8 computers respectively.

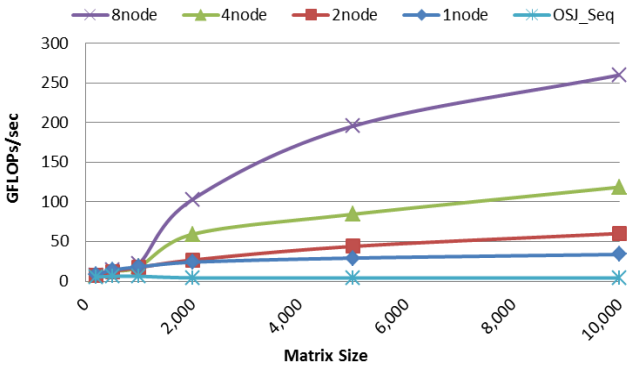
Obviously, the best performance can be achieved by combining all forms of parallel processing techniques (MPI, SIMD, and multi-threading). As Figure 13d shows, on large matrix size of 10,000×10,000, the performance in single-precision GFLOPS of HBJ improves from 73.98 GFLOPS on one node to 116.57, 255.76, and 515.62 GFLOPS on clusters with 2, 4, and 8 computers, respectively. As a result of this huge improvement in the performance, the speedups over the unparalleled OSJ algorithm are improved from 17.79 to 28.04, 61.52, and 124.03, respectively. While the speedups over the HBJ_Thrd_SIMD_1node are 1.58, 3.46, and 6.97 on 2, 4, and 8 computers, respectively.

7. CONCLUSION AND FUTURE WORK

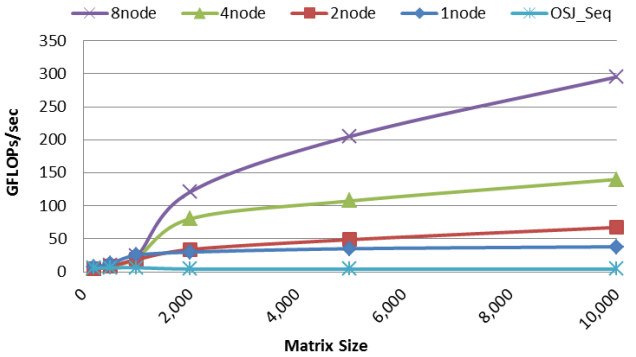
This paper discussed and evaluated the parallel implementations of applying Givens rotation (Level-1 BLAS), rank-1 update (Level-2 BLAS), and matrix multiplication (Level-3 BLAS) and the singular value decomposition algorithms on a cluster of Fujitsu Siemens CELSIUS R550 multi-core Intel processors. It applied parallel processing techniques to improve their performances by decreasing their execution times to run faster.



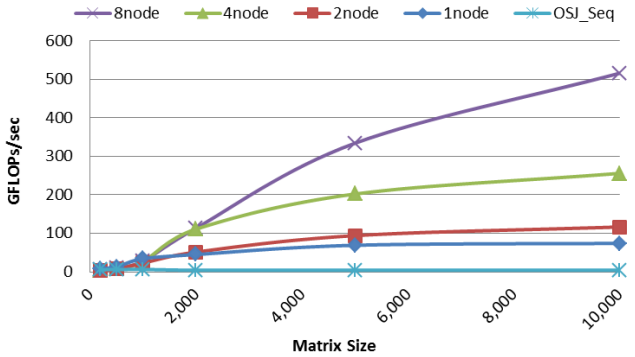
(a) Sequential performance



(b) SIMD Performance



(c) Multi-threading Performance



(d) Multi-threading SIMD Performance

Figure 13: Performance of the parallel HBJ on a cluster of 2, 4, and 8 nodes.

Table 1 summarizes the maximum performances in GFLOPS on one, two, four, and eight nodes. Applying Givens rotation (Level-1 BLAS) did not gain a performance due to using the multi-threading and MPI techniques because of the overhead of the threads creation and send/receive operations, respectively. However, they achieved a good performance when using the SIMD technique at large vector lengths that can fit in the cache. A performance of 8.08 GFLOPS was achieved for the apply Givens rotation by processing four elements using a single SIMD instruction. Therefore, a speedup of 3.84 over the sequential implementation could be achieved. It represented 96% from the ideal value, which equals four.

Moreover, using the multi-threading and MPI techniques did not improve the performance of the rank-1 update (Level-2 BLAS) because of the amount of data that each thread or each computer works on was still small. Besides, the number of floating-point operations was not enough to cancel the effect of the thread creation overhead or the network overhead. On the other hand, using the SIMD technique improved the performance of rank-1 update subroutine. The performance was enhanced to 3.93 GFLOPS, which achieved speedup of 3.54 over the sequential implementation. Furthermore, the performance improved by exploiting the memory hierarchy by reusing the loaded data in the cache memory many times by applying the blocking technique. The performance increased to 4.19 GFLOPS. Therefore, the speedup increased to 3.78 that represented 94.5% from the ideal.

Matrix-matrix multiplication (Level-3 BLAS) and singular value decomposition are important dense linear algebra algorithms, where exploiting all forms of parallelism improved their performance. On a single computer the performance of the traditional algorithm of the matrix-matrix multiplication achieved speedups of 3.76, 3.91, 5.40, 7.37, and 12.65 due to applying SIMD, multi-threading, SIMD-blocking, multi-threading SIMD, and multi-threading SIMD blocking techniques, respectively, over the sequential implementation on large matrix size (1000×1000) that can fit in the L2 cache. Furthermore, the performance improved when more than one computer are used for distributed processing. On ten computers at large matrix size of 10,000×10,000, the speedup of the traditional algorithm over the sequential implementation increased to 8.24, 13.34, 20.00, 37.73, and 104.04 when applying the MPI, SIMD MPI, multi-threading MPI, multi-threading-SIMD MPI, and multi-threading-SIMD-blocking MPI, respectively.

Finally, exploiting the SIMD, multi-threading, and multi-threading SIMD techniques speeded up the performances of the parallel block Jacobi (BJ) and hierarchal block Jacobi (HBJ) algorithms. For single-precision data, speedups of 3.73, 3.88, and 7.49 were achieved over the single threaded one-sided Jacobi (OSJ) algorithm for the block Jacobi algorithm at matrix size of 1000×1000 (the maximum size that can fit in the cache). However, for the HBJ algorithm the speedup increased to 8.10, 9.10, and 17.78 respectively, for single-precision data. Exploiting the capabilities of eight computers to execute the BJ and HBJ algorithms enhanced the performance furthermore. On matrix size 10,000×10,000 of single-precision data, the speedup of the BJ algorithm over the single

Table 1: Maximum performance (GFLOPS) on 1, 2, 4, and 8 node

	Sequential				SIMD				Multi-threading				Multi-threading SIMD				Multi-threading SIMD blocking			
	1	2	4	8	1	2	4	8	1	2	4	8	1	2	4	8	1	2	4	8
Apply Givens rotation	2.1	0.01	0.02	0.04	8.1	0.04	0.06	0.14	3.0	0.02	0.04	0.08	3.64	0.03	0.05	0.08	--	--	--	--
Rank-1 update	1.1	0.01	0.01	0.02	3.9	0.02	0.04	0.08	1.4	0.01	0.02	0.03	1.73	0.01	0.02	0.04	2.3	--	--	--
Matrix multiplication	1.9	1.8	3.5	6.7	7.2	2.6	5.0	9.8	7.5	4.1	7.9	15.7	14.07	7.2	14.3	28.7	24.2	17.5	37.58	79
One-sided Jacobi	6.2	--	--	--	23.3	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
Block Jacobi	6.3	7.5	15.8	31.2	23.2	27.5	55.0	101.0	23.6	24.9	54.1	107.0	45.6	50.5	103.3	207.0	--	--	--	--
Hierarchal block Jacobi	10.1	17.7	36.4	76.5	33.8	60.2	118.6	260.0	38.0	67.3	140	295.2	74.0	116.6	255.8	515.6	--	--	--	--

threaded OSJ increased to 7.50, 24.28, 25.73, and 49.79 for the MPI, SIMD MPI, multi-threading MPI, and multi-threading-SIMD MPI techniques, respectively. In addition, the speedup of the HBJ improved to 18.41, 62.54, 71.00, and 124.04 respectively.

Although we have discussed and evaluated parallel implementations of matrix-matrix multiplication and singular value decomposition algorithms that exploited the different forms of parallelism to improve their performance, a lot of work remains to be done. The following are some key areas for future research:

- Implement and evaluate the performance of dense linear algebra algorithms on advanced architectures like Intel Xeon Phi.
- Restructure dense linear algebra algorithms to exploit all forms of parallelism and memory hierarchy.
- Re-implement and evaluate the BJ and HBJ algorithms after replacing round-robin method by another to exploit memory hierarchy furthermore.
- Using the graphical processing units (GPUs) as another form of computer parallelism to improve the performance of the parallel algorithms on cluster of computers.
- Exploiting the cloud computing technology to execute and evaluate new implementations of parallel algorithms.

REFERENCES

[1] J. Hennessay and D. Patterson. *Computer Architecture A Quantitative Approach*, Morgan Kaufmann, 5th Edition, September 2011.

[2] Fujitsu, “data sheet of CELSIUS R550”, November 2009.

[3] Intel, *Intel® Xeon® Processor 5400 Series*, 2008.

[4] J. Ayala, M. López-Vallejo, and A. Veidenbaum, "Energy-efficient register renaming in high-performance processors," *Proceedings of WASP*, pp. 56-61, December 2003.

[5] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-way multithreaded SPARC processor," *Micro, IEEE*, Vol. 25, No. 2, pp. 21-29, 2005.

[6] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage," *ACM Transactions on Mathematical Software*, Vol. 5, No. 3, pp. 308-323, September 1979.

[7] J. Dongarra, J. Croz, S. Hammarling, and R. Hanson, "An Extended Set of Fortran Basic Linear Algebra Subprograms," *ACM Transactions on Mathematical Software*, Vol. 14, No. 1, pp. 1-17, March 1988.

[8] J. Dongarra, J. Croz, S. Hammarling, and I. Duff, "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Transactions on Mathematical Software*, Vol. 16, No. 1, pp. 1-17, March 1990.

[9] M. Soliman, "Performance Evaluation of Multi-Core Intel Xeon Processors on Basic Linear Algebra Subprograms", *Parallel Processing Letter (PPL), World Scientific Publishing Company*, ISSN: 0129-6264, Vol. 19, No. 1, pp. 159-174, March 2009.

[10] G. Golub and F. Luk, "Singular Value Decomposition: Applications and Computations," *Transactions of the Twenty-Second Conference of Army Mathematicians*, Vol. 577, pp. 577-605, 1977.

[11] G. Golub and C. Van Loan, *Matrix Computations*, John Hopkins University Press, Baltimore and London, 2nd edition, 1993.

[12] R. Brent and F. Luk, "The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays," *SIAM Journal on Scientific and Statistical Computing*, Vol. 6, No. 1, pp. 69-84, 1985.

[13] S. Rajasekaran and M. Song, "A relaxation scheme for increasing the parallelism in Jacobi-SVD," *Journal of Parallel and Distributed Computing*, Vol. 68, No. 6, pp. 769-777, 2008.

[14] M. Soliman, "Exploiting ILP, TLP, and DLP to Improve Multi-Core Performance of One-Sided Jacobi SVD," *Parallel Processing Letters, World Scientific Publishing Company*, Vol. 19, No. 2, pp. 355-375, March 2009.

[15] M. Soliman, "Memory hierarchy exploration for accelerating the parallel computation of SVDs," *Neural, Parallel & Scientific Computations*, Vol. 16, No. 4, pp. 543-561, 2008.

[16] M. Hestenes, "Inversion of matrices by biorthogonalization and related results," *Journal of the Society for Industrial and Applied Mathematics*, Vol. 6, No. 1, pp. 51-90, 1958.

[17] F. Van Zee, R. van de Geijn, and G. Quintana, "Restructuring the QR Algorithm for High-Performance Application of Givens Rotations," *FLAME Working Note #60*, October 2011.