# The Scalable Petascale Data-Driven Approach for the Cholesky Factorization with multiple GPUs

Yuki Tsujita
Tokyo Institute of Technology
Tokyo, Japan
tsujita.y.aa@m.titech.ac.jp

Toshio Endo
Tokyo Institute of Technology
& JST CREST
Tokyo, Japan
endo@is.titech.ac.jp

Katsuki Fujisawa
Kyushu University
& JST CREST
Fukuoka, Japan
fujisawa@imi.kyushu-u.ac.jp

## ABSTRACT

The Cholesky factorization is an important linear algebra kernel which is used in the semidefinite programming (SDP) problem. However, the large computation costs for Cholesky factorization of the Schur complement matrix (SCM) has been obstacles to solve large scale problems. This paper describes a brand-new version of the parallel SDP solver, SDPARA, which has been equipped with a Cholesky factorization implementation and demonstrated 1.7PFlops performance with over two million constraints by using 4,080 GPUs. The performance and scalability is even more improved by introducing a data-driven approach, rather than traditional synchronous approach. Also we point out that typical data-driven implementations have limitation in scalability, and demonstrate the efficiency of the proposed approach via experiments on TSUBAME2.5 supercomputer.

## Keywords

GPGPU, Dense Matrix Algebra, Semidefinite Programming, Data-Driven

## 1. INTRODUCTION

The semidefinite programming (SDP), which involves a positive semidefinite symmetric matrix variable and/or a linear matrix inequality as its constraints, is not merely an extension of linear programming (LP) to the space of symmetric matrices but has played a significant role in the field of mathematical optimization programming(MOP) for more than two decades due to its wide-range applications such as nonconvex optimization[1], combinatorial optimization[2], structural optimization[3], quantum chemistry[6, 5], sensor network location, and machine learning. The semidefinite programming (SDP) problem is a predominant problem in mathematical optimization. The primal-dual interior-point method (PDIPM)[13, 12, 14, 16, 17] is one of the most powerful algorithms for solving SDP problems, and many research groups have employed it for the development of software packages. The SDPA (SemiDefinite Programming Algorithm) [13], one of the first SDP software packages which implement PDIPM, has been developed and maintained by Fujisawa et al. over two decades. As applications of SDPs broadened, we need high performance for larger scale SDPs. SDPARA (SemiDefinite Programming Algorithm paRAllel version) [11, 9] is a parallel version of SDPA, which runs on multiple CPU cores and/or processors and shared and/or distributed memory environment with MPI [1] .

One of bottleneck routines of SDPARA is called CHOLESKY, which corresponds to the Cholesky factorization of the Schur complement matrix (SCM). This routine has traditionally used a parallel linear algebra library, ScaLAPACK[2]. More recently, we have developed a GPU version [7, 18], to largely accelerate the CHOLESKY routine based on technologies in [25]. This version successfully solved the largest SDP problem (which has over 2.33 million constraints), creating a new world record; the performance of CHOLESKY was 1.713 PetaFlops in double precision [18], using 2,720 CPUs and 4,080 GPUs on the TSUBAME2.5 supercomputer at Tokyo Institute of Technology.

The objective of this paper is further improvement of the above petascale implementation. While it has been already scalable, we found that there are still rooms to reduce the amount of PCIe communication between host and device. Also since the computations are synchronous, the performance tends to be affected by the MPI costs largely.

In order to improve this situation, we integrate the data driven approach, where the tasks are divided into block/tile level[19, 23]. This approach has been considered promising, since it can overlap computation and communication in asynchronous ways. Also reduction of PCIe communication amount is possible by maintaining memory coherency in a tile-wise way[22].

However, even up-to-date data driven implementations still have latent bottlenecks on larger scale environments, such as the TSUBAME2.5 supercomputer, which has more than 1,000 computing nodes. This paper proposes techniques to improve scalability, which are:

[1]SDPA and its variants can be obtained under the GPL license from the following web site: http://sdpa.sourceforge.net/
[2]http://www.netlib.org/scalapack/

- A scalable data transfer method in solving data dependency, which is especially important when the dependency graph includes nodes with large (dozen of) fan-out
- A scalable termination detection method

Through the performance comparison, we demonstrate the optimized implementation achieves 27% performance improvement over the previous ones.

## 2. PDIPM

This section briefly describes the primal-dual interior-point method (PDIPM) and that Cholesky factorization is time-consuming part of it. For detailed description, refer to previous papers [11, 12, 13, 14, 16, 17].

We regard $R^{n\times n}$ as $n^2$-dimensional Euclidean space. Let $\mathcal{S}^n$ denote the set of all $n \times n$ symmetric real matrices; $\mathcal{S}^n$ forms an $n(n+1)/2$-dimensional linear subspace of $R^{n\times n}$. For each pair of $\boldsymbol{X}$ and $\boldsymbol{Z}$ in $R^{n\times n}$, $\boldsymbol{X} \bullet \boldsymbol{Z}$ stands for the inner product of $\boldsymbol{X}$ and $\boldsymbol{Z}$, i.e., Tr $\boldsymbol{X}^T\boldsymbol{Z}$, the trace of $\boldsymbol{X}^T\boldsymbol{Z}$. We write $\boldsymbol{X} \succ \boldsymbol{O}$ if $\boldsymbol{X} \in \mathcal{S}^n$ is positive definite, and $\boldsymbol{X} \succeq \boldsymbol{O}$ if $\boldsymbol{X} \in \mathcal{S}^n$ is positive semidefinite. Here $\boldsymbol{O}$ denotes the $n \times n$ zero matrix.

Let $\boldsymbol{C} \in \mathcal{S}^n$, $\boldsymbol{A}_i \in \mathcal{S}^n$ $(1 \le i \le m)$ and $b_i \in R$ $(1 \le i \le m)$. Consider the semidefinite program and its dual:

$$\left.\begin{array}{lll} \mathcal{P}: & \text{minimize} & \boldsymbol{C} \bullet \boldsymbol{X} \\ & \text{subject to} & \boldsymbol{A}_i \bullet \boldsymbol{X} = b_i \ (1 \le i \le m), \ \boldsymbol{X} \succeq \boldsymbol{O}. \\ \mathcal{D}: & \text{maximize} & \displaystyle\sum_{i=1}^{m} b_i y_i \\ & \text{subject to} & \displaystyle\sum_{i=1}^{m} \boldsymbol{A}_i y_i + \boldsymbol{Z} = \boldsymbol{C}, \ \boldsymbol{Z} \succeq \boldsymbol{O}. \end{array}\right\} \tag{1}$$

We assume that the set of $n \times n$ symmetric matrices $\boldsymbol{A}_i$ $(1 \le i \le m)$ is linearly independent. This implies that $m \le n(n+1)/2$. We say that $(\boldsymbol{X}, \boldsymbol{y}, \boldsymbol{Z})$ is a feasible solution (an interior-feasible solution, or an optimal solution, respectively.) of the SDP (1) if $\boldsymbol{X}$ is a feasible solution (an interior-feasible solution, i.e., a feasible solution satisfying $\boldsymbol{X} \succ \boldsymbol{O}$, or a minimizing solution, respectively) of $\mathcal{P}$ and $(\boldsymbol{y}, \boldsymbol{Z})$ is a feasible solution (an interior-feasible solution, i.e., a feasible solution satisfying $\boldsymbol{Z} \succ \boldsymbol{O}$, or a maximizing solution, respectively) of $\mathcal{D}$.

Next, we show a generic primal-dual interior-point method for the SDP, on which SDPA and SDPARA are based.

Step 0: Determine a stopping criterion, and choose an feasible or infeasible initial point $(\boldsymbol{X}^0, \boldsymbol{y}^0, \boldsymbol{Z}^0)$ such that $\boldsymbol{X}^0 \succ \boldsymbol{O}$ and $\boldsymbol{Z}^0 \succ \boldsymbol{O}$. Let $(\boldsymbol{X}, \boldsymbol{y}, \boldsymbol{Z}) = (\boldsymbol{X}^0, \boldsymbol{y}^0, \boldsymbol{Z}^0)$.

Step 1: If the current point $(\boldsymbol{X}, \boldsymbol{y}, \boldsymbol{Z})$ satisfies the stopping criterion, output it and stop the iteration.

Step 2: Choose a search direction $(d\boldsymbol{X}, d\boldsymbol{y}, d\boldsymbol{Z})$.

Step 3: Choose a primal step length $\alpha_p$ and a dual step length $\alpha_d$ such that

$$\boldsymbol{X} + \alpha_p d\boldsymbol{X} \succ \boldsymbol{O} \ \text{ and } \ \boldsymbol{Z} + \alpha_d d\boldsymbol{Z} \succ \boldsymbol{O}.$$

Let

$$\boldsymbol{X} = \boldsymbol{X} + \alpha_p d\boldsymbol{X} \ \text{ and } \ (\boldsymbol{y}, \boldsymbol{Z}) = (\boldsymbol{y}, \boldsymbol{Z}) + \alpha_d(d\boldsymbol{y}, d\boldsymbol{Z}).$$

Step 4: Go to Step 1.

In SDPA and SDPARA, the most time consuming part is Step 2, which is further divided into the ELEMENTS routine, where all elements of SCM is computed, and the CHOLESKY routine, where we obtain the Cholesky factorization of SCM. The time complexities of ELEMENTS and CHOLESKY by $O(mn^3+m^2n^2)$ and $O(m^3)$, respectively, where $n$ and $m$ are defined as follows, (1) $n$: the size of the variable matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$, and (2) $m$: the number of equality constraints in the dual form $\mathcal{D}$, which equals the size of the Schur Complement Matrix (SCM). Among of two routines, the costs of CHOLESKY, the target for improvement in this paper, tend to dominate if the SCM is dense[7].

Table 1 shows the performance record of CHOLESKY of SDPARA. In 2003 [11], we have released the SDPARA 1.0.1 and achieved 78.58GFlops. Our implementation with GPUs achieved 1.713 PFlops for large-scale Cholesky factorization using 4,080 GPUs in 2014 [18].

Table 1: Performance record of CHOLESKY of SDPARA

| Year | Paper | $n$ | $m$ | CHOLESY (Flops) |
|---|---|---|---|---|
| 2003 | [11] | 630 | 24,503 | 78.58 Giga |
| 2010 | [4] | 10,462 | 76,554 | 2.414 Tera |
| 2012 | [7] | 1,779,204 | 1,484,406 | 0.533 Peta |
| 2014 | [18] | 2,752,649 | 2,339,331 | 1.713 Peta |

## 3. BACKGROUND AND MOTIVATION

### 3.1 GPGPU and TSUBAME2.5 Supercomputer

GPGPU(General Purpose Graphics Processing Unit) is a technique to use computing resources of GPU (Graphics Processing Unit) for a general-purpose calculation as well as image processing. Compared with CPUs, GPUs are designed to make throughput of computation higher; thus, they have been successful in parallel computations with regular structures, including matrix operations. In this paper, we use CUDA programming environment designed for NVIDIA GPUs, however, the proposed techniques are applicable to other environments. Our new implementation is evaluated on the TSUBAME2.5 GPGPU petascale supercomputer at Tokyo Institute of Technology, though SDPARA and the proposed techniques were developed for general GPU supercomputers and clusters. Also the approach is applicable to systems with Intel Xeon Phi, though we need to replace accelerated BLAS kernels.

The main part of TSUBAME2.5 consists of 1408 HP Proliant SL390s G7 computing nodes. Figure 1 shows the structure of each node, which has two Intel Xeon X5670 2.93 GHz (six cores) CPUs, three NVIDIA Tesla K20X GPUs, 54 GB (partly 96 GB) of DDR3 memory. Each node is connected to interconnect via two QDR 40 Gbps InfiniBand HCAs. Each K20X GPU, which we mainly use for kernel computations, has the peak performance of 1.31TFlops. The peak performance of TSUBAME2.5 system is 5.76PFlops, including $3 \times 1,408$ K20X GPUs.

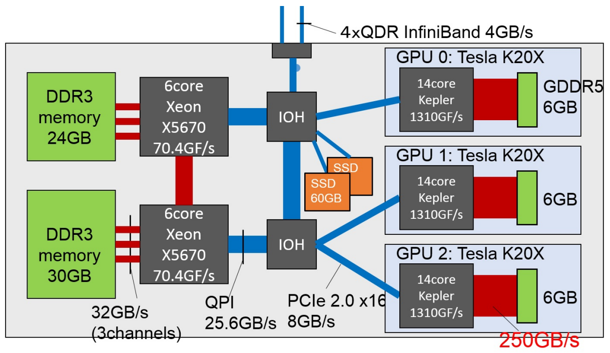

Figure 1: Structure of each HP SL390s G7 node used in TSUBAME 2.5 with Intel Xeon CPUs and three NVIDIA K20X GPUs.

While GPUs have higher computation throughput and memory bandwidth, they have limitations on memory size. The device memory capacity is 6GB per GPU. while the host memory can be expanded more easily (54GB on the TSUBAME nodes). Therefore in order to support larger scale computation, we should harness the capacity of host memory. However, we should consider costs of data movement between CPUs and GPUs (hereafter we call it PCIe communication). Since the bandwidth of PCI-Express (PCIe), 8GB/s in our case, is much smaller than device memory bandwidth (250GB/s on K20X), we have to reduce the amount of PCIe communication for better performance.

## 3.2 Cholesky factorization and Parallel Implementations

The Cholesky factorization takes a symmetric positive definite matrix $A$, and outputs a lower triangle matrix $L$, where $A = LL^T$ [3]. We assume $A$ is a dense matrix whose size is $m \times m$.

The Cholesky factorization is a well known computation and its parallel implementation has a long history. The most well-known one is included in the ScaLAPACK parallel linear algebra library[24]. Here the matrix $A$ is divided into blocks with a uniform size $n_b \times n_b$, which are distributed among processes in two-dimensional block cyclic method. The previous version of SDPARA used a multi-GPU implementation based on this algorithm, though it is enhanced with a communication overlapping method[7, 18]. The ScaLAPACK approach and its variants are used for long, however, some drawbacks have been reported.

- The computation is done in a synchronous style; all blocks owned by a single process are basically updated at once. Thus the total performance tends to be heavily affected by inter-node communication costs.
- When the routine is implemented for GPUs, it is harder to reduce PCIe communication. Our previous implementation has assumed that all the matrix data is available on host memory after each kernel finishes. We could reduce PCIe communication if the matrix data can reside in device memory over several iterations of the outer loop.

[3]Although this section uses a denotation of $A$, it differs from $\boldsymbol{A}_i$ in the previous section

To overcome such drawbacks, several research groups adopt the data driven approach[20, 23, 22]. Here, we change the data format for the matrix; instead of the single rectangular format, we let each process maintain several blocks (or tiles), each of which is an array of $n_b \times n_b$ size.

Then the entire computation is done in an asynchronous style. The Cholesky factorization computation is broken down into the tile level, and each divided computation is called a task. While we need to consider dependency among tasks as shown in Figure2, such division enables to harness more parallelism. In distributed environments, one task may depend on some data that are produced by remote processes. In such cases, inter-process communication is involved.

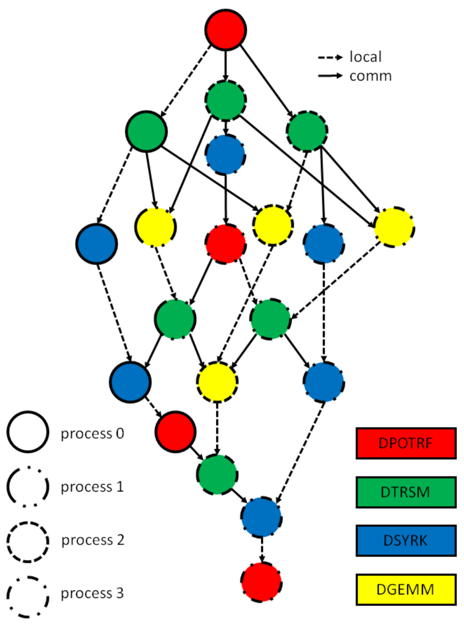

Figure 2: Direct Acyclic Graph(DAG) of the Cholesky factorization

On GPGPU systems, we additionally have to consider the memory hierarchy of device memory and host memory [21, 22]. If the input block is not available on the device memory, PCIe communication is involved. After all the input blocks are available on the device memory, we can execute the task. If the device memory is already full, some blocks are swapped out to the host memory.

With this asynchronous, data-driven approach, it is known that the performance is improved, since computation, MPI communication and PCIe communication are naturally overlapped. However, it is unproven whether it is scalable up to hundreds or thousands of nodes (as far as we know, DAGuE and its successor, PaRSEC [23] are evaluated on 64nodes or less).

## 3.3 Motivation for Scalable Data Driven Execution

We consider following elements with the data-driven implementations can become latent bottlenecks on larger scale environments.

- Generally, data of a single tile may be consumed by several processes. In Cholesky factorization, tile data may used by $(P+Q)$ processes, where $(P \times Q)$ corresponds to the size of process grid. If a single source process sends data to the consumer process one by one, the bandwidth of the source node becomes bottleneck. In this point, the traditional implementation such as Scalapack might have advantages, since this problem has been avoided by using scalable MPI group communication, such as MPI_Bcast. In data-driven approach, each consumer process may request the data to source asynchronously, thus MPI_Bcast cannot be used.

- Next, termination detection of each process is not trivial. In data-driven implementations, each MPI process can act both as data servers and clients. This property makes a scalable implementation harder; a single process cannot exit the computation loop, even after all of its local tasks have been finished. Thus a scalable termination detection method is required.

## 4. DATA DRIVEN CHOLESKY FACTORIZATION

This section briefly explains our data-driven implementation of the Cholesky factorization for GPGPU supercomputers. For more details, refer to [22]. After describing the basic data driven scheduling method, we improve its scalability in the next section.

### 4.1 Basic Implementation

We divide the input matrix data $A$ into the units called "tiles", each of which has $n_b \times n_b$ size. The tiles are distributed among MPI processes in a two-dimensional block cyclic style. Instead of holding all the tiles included in $A$, we hold only tiles for the lower triangular part of $A$ , because Cholesky factorization assumes $A$ as a symmetric matrix. When the computation is started, the tile data is put on the host memory.

Then each MPI process updates its local tiles in an asynchronous style, conforming to the task dependency shown in Figure 2. In order to maintain the task dependency, we let each tile have additional variables as follows. First, each tile maintains a variable to express its current running step; unlike synchronous implementations, where each process has a single loop iteration variables, each tile needs its step.

In our implementation, each MPI process consists of several (two or three typically) worker threads and a ignition thread. Relationships among threads and processes are illustrated in Figure 3 (for simplicity, a process is shown with only one worker thread). Each process has its task queue, shared by all its threads, in order to manage the runnable tasks on the process.

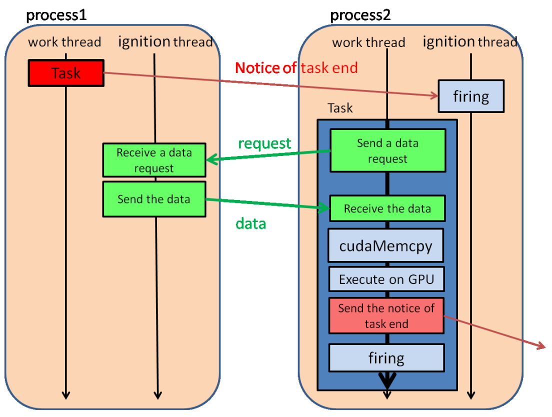

Figure 3: Threads and processes in our implementation. The figure also shows MPI Communication pattern when a task is finalized

Each worker thread repeats the following steps, task select, localize, execute, and finalize, continuously.

Task select It takes out a runnable task from the task queue if exists; we let $T$ be the target tile of the task. If the task queue is empty, the calculation is blocked.

Localize Generally, execution of a task requires the output data of the precedent tasks as inputs. We let $T_{i1}, T_{i2}$ be the result tiles of the precedent tasks [4]. Then the worker thread checks the state of tiles $T, T_{i1}, T_{i2}$ and executes the corresponding operations as follows.

1. if the tile data is on device memory, nothing is required.
2. if the tile data is not on device memory, but on the local host memory, the tile data is copied to device memory via PCIe bus. This may involve swapping out operation, as described below.
3. if the tile data is neither on device memory nor on local host memory, the worker thread sends a request message to the owner process, and issues MPI_Recv in order to receive the tile data. After the data arrival, we execute as in Case 2.

Execute Now all the required tile data are available on GPU; thus we execute the calculation task. The task is one of DPOTRF, DTRSM, DSYRK or DGEMM, according to the state of tile (Figure 2). It typically involves invocation of a BLAS function on GPU. However, since DPOTRF derives few benefits from GPU, it is executed on CPU.

Finalize When a task $A$ is finished, the worker thread performs operations for the following tasks, which require the output of this task as shown in Figure 3. For this purpose, the worker thread sends notice messages to all processes that have following tasks of task $A$. These messages are handled by receivers' ignition thread as described below.

[4] In Cholesky factorization, each task depends on two tasks or less.

We introduce multiple worker threads in order to achieve overlapping of calculation, PCIe communication and MPI communication in a simple implementation.

On the other hand the ignition thread continuously polls request messages and notice messages from other processes, and handles them as described in [22].

## 4.2 Memory Management

In our implementation, each process put data of all the tiles owned by the process on the host memory. On the other hand, the smaller GPU device memory is used like a "cache" of the host memory; currently we use a simple coherency protocol similar to the well-known MSI protocol. When a process copies a tile data to GPU, it needs to evict another tile if the device memory is full. Then one of residing tile on GPU is selected as a victim, and copied back if its content is newer than data on host memory (the data is DIRTY). For selecting a victim tile, there are several possible strategies as described in [22]. According to the preliminary experiment, we use FIFO strategy in this paper.

## 4.3 Task Selection

As previously described, we manage the runnable tasks by using the task queue per process. Since a task queue may contain several runnable tasks, we need to make strategies to select a task to be executed. After comparison of four strategies, FIFO, Random, Greed, ByIJ [22], we selected the Random strategy, where we take one of runnable tasks randomly since it empirically shows good result.

## 5. SCALABILITY IMPROVEMENT

## 5.1 Scalable Data Transfer

As described in Section 3.3, in data-driven Cholesky factorization, each tile may be consumed by $P+Q$ processes, where $P$ is the number of process row and $Q$ is the number of process column. On TSUBAME2.5 supercomputer, the size of process grid may be up to $68 \times 60$, thus sequential sending of the same tile data for $68+60$ times by a single source process degrades the total performance. In order to avoid this bottleneck, we did not use MPI-3 non-blocking collective communication such as MPI_Ibcast, since in data-driven execution, the order of incoming messages are indeterministic. Instead, we implemented a scalable data transfer method based on the tree topology communication. Similar idea is more popular in peer-to-peer area; thus convergence of HPC technology and P2P technology become more important as the supercomputers become larger towards Exascale.

For this purpose, we let processes maintain an additional list structure per tile, called CSlist (client and server list). The example is shown in Figure 4.

Step (1): We assume that Process 1 finished a task on Tile 1, and knows all the processes (Processes 2 to 5) that consume the tile. Then Process 1 prepares a CSlist for Tile 1 as in Step (1) in the figure. Each column corresponds to a consumer process, whose "server" is Process 1 itself. We assume that Process 1 received a request message from Process 3 first.

Step (2): Since Process 1 received a request message, it has to send data of Tile 1 to Process 3. Also it performs additional tasks as follows. Process 1 updates its CSlist so that partial clients (for example, half clients that Process 1 knows) are directed to the current requester (Process 3) as the server. In the figure, CSlist on Process 1 has a column {Client=2, Server=3}. In addition to the tile data, Process 1 also sends the partial CSlist, related to Process 3, to it. Now Process 3 can work as a delegated data server.

Step (3): Next, we assume Process 1 received a request message from Process 2 (note that other consumers do not know the existence of delegated servers). Process 1 checks the CSlist, and finds that the corresponding server for Process 2 is Process 3. Thus it forwards the request message to Process 3, instead of sending data directly. When Process 3 receives the request message, it sends data of Tile 1 to the original requester, Process 2.

The delegation/forwarding may occur in a recursive style, if we have more consumers. This implementation tends to make the hop count increase up to $O(log_2(P+Q))$; however, as shown in the next section, we surrender the advantage of reducing the concentration of MPI communications.

This method arises another issue of 'garbage collection'. In the figure, when can Process 3 discard the cache of Tile 1? In order to determine it, the ignition thread (receiver of request messages) updates CSlist; if it handles the request as a data server, it removes the original requester from the CSlist. A process can discard the cache of a tile if (a) its CSlist is empty and (b) all the local consumer tasks in the process have been finished.

## 5.2 Scalable Termination Detection

Another issue is termination detection of the entire Cholesky factorization. A process cannot exit even after all of its local tasks have been finished, since it may still receive requests for its data from other running processes.

As a first step, we implemented a simple method as follows. If a process finishes all local tasks, it sends finished message to all other processes. Each process counts the finished messages while polling possible request messages, until it receives finished messages from all other processes. This simple implementation suffers from bottleneck, since we need $O(P \times Q)^2$ messages in the situation supposed in Section 5.1.

Scalability is improved by using the CSlist described in previous section. Each process maintains information about 'which clients have been requested this tile, and who have not yet' in the CSlist. Thus for arbitrary process, it will not receive further request messages when CSlists for all local tiles (including cached tiles) become empty. Thus we can do termination detection in a scalable style.

## 6. PERFORMANCE EVALUATION

This section describes the performance evaluation results of the new version CHOLESKY for large-scale SDP problems on TSUBAME2.5 supercomputer; the system software

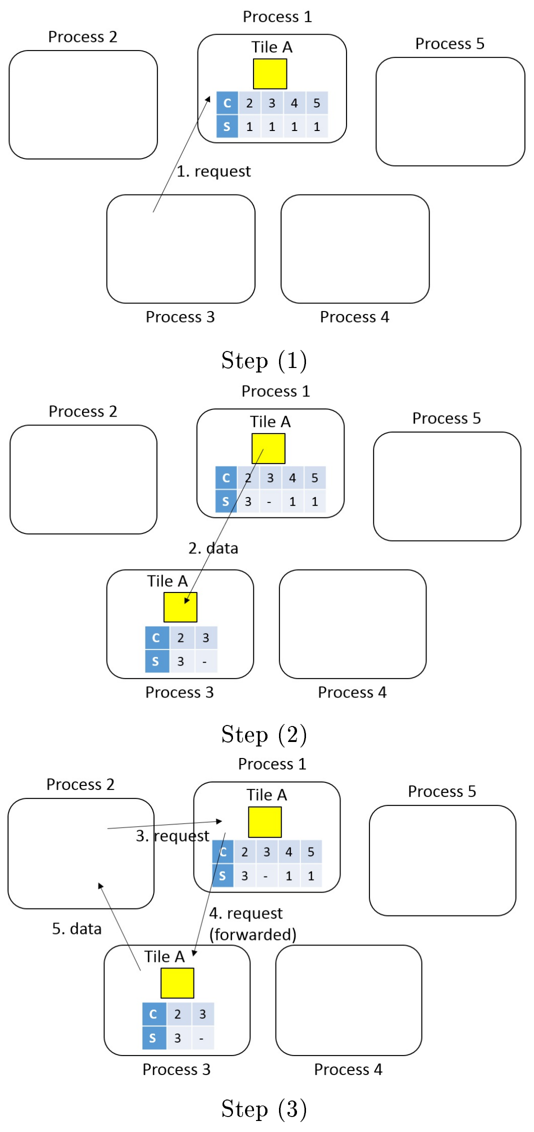

Figure 4: Scalable tree-type data transfer algorithm

configuration is shown in Table 2. In the evaluation, three GPUs per node are used for kernel computation, and each GPU is mapped to a single MPI process. Each MPI process consists of one ignition thread and four worker threads. As the size of each tile, we choose $n_b = 2048$ according to the results of preliminary experiments.

SDP problems used in the evaluation are listed in Table 3. Among them, the problem that produces largest matrix is QAP10, where the SCM size is $m = 2,339,331$, which occupies about 20TB in the triangular form.

Figure 5 shows the speed of CHOLESKY component by using up to 400 nodes (1200 GPUs). It compares three versions of CHOLESKY:

- 2014: the synchronized implementation in SDPARA 7.6.0-G[18]

Table 2: System configuration used in the experimentation

| | |
|---|---|
| Compiler | gcc 4.3.4 |
| MPI | MVAPICH 2.0rc1 (with support of MPI_THREAD_MULTIPLE) |
| CUDA | 6.5 |
| BLAS (GPU) | CUBLAS 6.5 |
| BLAS (CPU) | MKL 2013.1 |

Table 3: Problems used in the evaluation of CHOLESKY

| Name | SCM size ($= m$) |
|---|---|
| QAP5 | 379,350 |
| QAP6 | 709,275 |
| QAP7 | 1,218,400 |
| QAP8 | 1,484,406 |
| QAP10 | 2,339,331 |

- No-TREE: data-driven execution is introduced, but tree-type communication is not introduced.
- Latest: our latest version with data-driven execution, tree-type communication and scalable termination detection.

According to the results, the performance of the latest version reaches 642 TFlops with QAP7 problem by using 400 nodes (1200GPUs). Due to the property of Cholesky factorization that computation complexity is $O(m^3)$ and communication complexity is $O(m^2)$, it is natural that larger problems enjoy better performance. When we compare three versions, Latest achieve the best performance in all cases. It shows 27% improvement over the 2014 version, for QAP7 problem with 400 nodes. On the other hand, the scalability of No-TREE version is much worse, and it is even slower than the old 2014 version. These results indicate the fact that scalable group communication method is mandatory for data-driven schedulers to obtain good scalability with hundreds nodes or more.

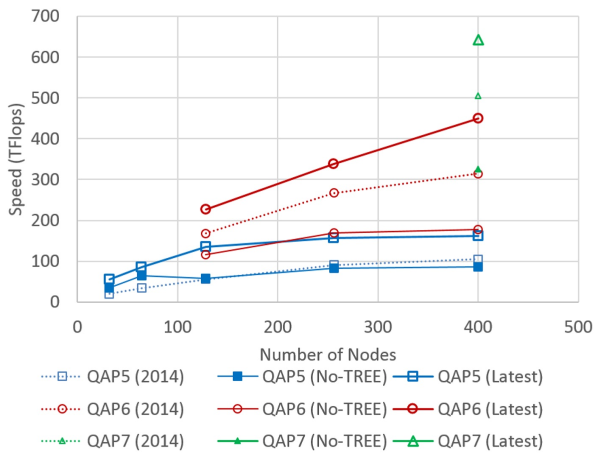

Figure 5: Performance of GPU CHOLESKY obtained by using up to 400 nodes (1,200 GPUs) on TSUBAME2.5.

Results with larger scale are shown in Figure 6, with up to 1,360 nodes (4,080 GPUs), almost the whole system of TSUBAME2.5. Unfortunately, due to the limited time when we were allowed to use the whole system, we could not execute the latest version in this experiment. While the "With-TREE" version is equipped with the tree-type communication, it does not have scalable termination detection, unlike Latest version.

The graph shows that the "With-TREE" version achieves 1.510 PFlops for QAP10 problem with 1,360 nodes. This is 12% lower than 1.713 PFlops of 2014 version, thus we could not renew our previous world record. Also we observe the scalability of "With-TREE" is compromised with 700 nodes or more. We need to investigate the reason for this; currently we consider this is due to lack of scalable termination detection as mentioned above. If we executed the Latest version on the whole system, considering the performance ratio between Latest and 2014 with 400nodes, we expect the performance around $1.713 \times 1.27 = 2.175$ PFlops.

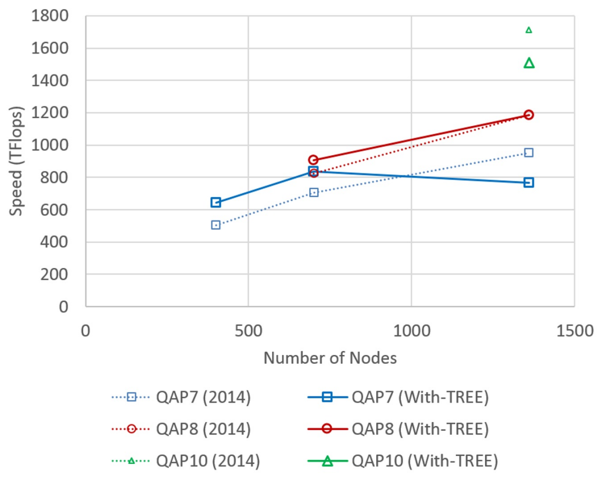

Figure 6: Performance of GPU CHOLESKY obtained by using up to 1,360 nodes (4,080 GPUs) on TSUBAME2.5.

## 7. RELATED WORK

Our data driven scheduling method is strongly influenced by DAGuE/PaRSEC by Bosilca et al. [19, 20, 23]. They have presented a direct acyclic graph (DAG) scheduler for distributed environments with GPUs, and demonstrated that the scheduler can execute applications including the Cholesky factorization efficiently. For the scalability issue, they have mentioned the possibility of introducing efficient collective communication methods [20]; however, to our knowledge, its integration and/or evaluation have not been published. In this paper, we demonstrated that this issue is the key to determine the scalability of implementation when we use hundreds or thousands nodes. Also we could embed our scalable data transfer and termination detection methods in their implementation.

StarPU[21] is a DAG scheduling framework for heterogeneous environments. It allows for each task to run either on CPUs or GPUs according to the resource utilization, in order to improve the performance of execution of the whole task graph. It also maintains data consistency, while mitigating data movement between CPUs and GPUs. However, StarPU does not have scalability improvement techniques as described in this paper.

In order to harness memory hierarchy of GPU memory and CPU memory in a transparent style, authors have proposed a runtime library called hybrid hierarchical runtime (HHRT)[26]. HHRT uses an oversubscription model; each GPU is shared by multiple processes, and when GPU memory is full, data of some processes are automatically swapped out. This methodology is successful for stencil based applications, however, we did not adopt it for the Cholesky factorization. One of the reasons is that using MPI communication between processes on the same node degrades the overall performance for this computation. Also the memory consumption would be increased because of the lack of the mechanism for sharing memory objects among processes. After these problems are solved, we could integrate HHRT and the scheduling methods in this paper.

## 8. CONCLUSION AND FUTURE WORK

We have described a scalable data driven implementation for the optimization of the multi-node multi-GPU Cholesky factorization, which is the most important kernel in SDPARA, the petascale SDP problem solver. It solves the scalability issues in typical data driven implementation by introducing scalable data transfer method and termination detection method. In spite of these advantages, we do not spoil advantages of data driven approaches; the communication amount between CPU and GPU is reduced, and computation and communication are overlapped. Compared with the synchronous implementation, our implementation achieved 27% performance improvement on 400 nodes and 1,200 GPUs of TSUBAME2.5 supercomputer. Due to limited time for experiments, the latest version cannot executed with the entire system, however, we expect its performance will be more than 2 PFlops.

In future, we will prove the scalability of the latest implementation with more than 1,000 nodes. Also we have discovered that memory consumption of each process is indeterministic, which may incur host memory overflow. We will investigate relationship between task selection strategies and changes of memory consumption. Also we will extend the implementation so that it can harness non-volatile memory while harnessing deeper memory hierarchy efficiently, towards extreme scale SDP problems.

## 9. REFERENCES

[1] A. Takeda, K. Fujisawa, Y. Fukaya and M. Kojima, Parallel implementation of successive convex relaxation methods for quadratic optimization problems, Journal of Global Optimization, Vol. 24, No. 2, pp237–260, 2002.

[2] K. Fujisawa, M. Fukuda, M. Kojima and N. Nakata, Numerical evaluation of the SDPA (SemiDefinite Programming Algorithm), The High Performance Optimization, Kluwer Academic Publishers, pp 267-301, 1999.

[3] M. Ohsaki, K. Fujisawa, N. Katoh and K. Kanno, Semi-definite programming for topology optimization of truss under multiple eigenvalue constraints, The Computer Methods in Applied Mechanics and Engineering, Vol. 180, pp 203-217, 1999.

[4] M. Nakata, M. Fukuda and K. Fujisawa, Variational Approach to Electronic Structure Calculations on Second-Order Reduced Density Matrices and the $N$-Representability Problem, H. Siedentop (eds.), Complex Quantum Systems - Analysis of Large Coulomb Systems, Institute of Mathematical Sciences, National University of Singapore, pp163-194, 2013.

[5] M. Nakata, B. J. Braams, K. Fujisawa, M. Fukuda, J. K. Percus, M. Yamashita and Z. Zhao, Variational calculation of second-order reduced density matrices by strong N-representability conditions and an accurate semidefinite programming solver, The Journal of Chemical Physics, 128, 164113, 2008.

[6] J. S. M. Anderson, M. Nakata, R. Igarashi, K. Fujisawa and M. Yamashita, The second-order reduced density matrix method and the two-dimensional Hubbard model, Computational and Theoretical Chemistry, Available online 23, August 2012.

[7] K. Fujisawa, T. Endo, H. Sato, M. Yamashita, S. Matsuoka and M. Nakata: High-performance general solver for extremely large-scale semidefinite programming problems, Proceedings of the 2012 ACM/IEEE Conference on Supercomputing, SC'12, (2012)

[8] M. Yamashita, K. Fujisawa, M. Fukuda, K. Nakata and M. Nakata, Parallel solver for semidefinite programming problem having sparse Schur complement matrix, the ACM Transactions on Mathematical Software, Volume 39, Number 12, 2012.

[9] K. Fujisawa, K. Nakata, M. Yamashita and M. Fukuda, SDPA Project : Solving Large-scale Semidefinite Programs, Journal of the Operations Research Society of Japan, Vol.50, No.4, pp278-298, 2007.

[10] K. Nakata, M. Yamashita, K. Fujisawa and M. Kojima, A Parallel Primal-Dual Interior-Point Method for Semidefinite Programs Using Positive Definite Matrix Completion, Journal of Parallel Computing,Vol.32, pp24-43 2006.

[11] M. Yamashita, K. Fujisawa and M. Kojima, SDPARA : SemiDefinite Programming Algorithm PARAllel Version, Journal of Parallel Computing, Vol 29/8, pp1053–1067, 2003

[12] K. Nakata, K. Fujisawa, M. Fukuda, M. Kojima and K. Murota, Exploit sparsity in semidefinite programming via matrix completion II: Implementation and numerical results, Mathematical Programming, Ser.B, Vol 95, pp303–327, 2003.

[13] K. Fujisawa, M. Kojima and K. Nakata, Exploiting sparsity in primal-dual interior-point methods for semidefinite programming, Mathematical Programming, Vol. 79, pp 235-253, 1997.

[14] B. Borchers: CSDP 2.3 user's guide. Optimization Methods and Software, 11/12 (1999), 597–611. Available from http://www.coin-or.org/projects/Csdp.xml.

[15] K. Fujisawa, M. Fukuda, K. Kobayashi, M. Kojima, K. Nakata, M. Nakata, and M. Yamashita: SDPA (SemiDefinite Programming Algorithm) User's Manual — Version 7.0.5, Research Report B-448, Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, 2008.

[16] J. F. Sturm: Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones. Optimization Methods and Software, 11/12 (1999), 625–653. Available from http://sedumi.ie.lehigh.edu/.

[17] K.-C. Toh, M. J. Todd, and R. H. Tüüncü: SDPT3 — a MATLAB software package for semidefinite programming, version 1.3. Optimization Methods and Software, 11/12 (1999), 545–581. Available from http://www.math.nus.edu.sg/∼mattohkc/sdpt3.html.

[18] K. Fujisawa, T. Endo, Y. Yasui, H. Sato, N. Matsuzawa, S. Matsuoka and H. Waki: Peta-scale general solver for semidefinite programming problems with over two million constraints, In Proceedings of IEEE International Parallel & Distributed Processing Symposium (IPDPS), (2014)

[19] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. DAGuE: A generic distributed dag engine for high performance computing. Technical Report ICL-UT-10-01, Innovative Computing Laboratory, (2010)

[20] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. DAGuE: A generic distributed dag engine for high performance computing. Parallel Computing, volume 38, pages 27-51 (2012)

[21] Cedric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-Andre Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience, pages 187-198, (2011)

[22] Yuki Tsujita and Toshio Endo. Data driven scheduling approach for the multi-node multi-GPU Cholesky decomposition. In Proceedings of Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), in conjunction with IEEE IPDPS 2015 (2015)

[23] Wei Wu, Aurelien Bouteiller, George Bosilca, Mathieu Faverge, Jack Dongarra. Hierarchical DAG Scheduling for Hybrid Distributed Systems. In Proceedings of IEEE International Parallel & Distributed Processing Symposium (IPDPS), (2015) (to appear)

[24] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley. The design and implementation of the SCALAPACK LU, QR, and Cholesky factorization routines. Technial Report UT CS-94-246, LAPACK Working Note 80, (1994)

[25] Toshio Endo, Akira Nukada, Satoshi Matsuoka and Naoya Maruyama. Linpack Evaluation on a Supercomputer with Heterogeneous Accelerators. In Proceedings of IEEE International Parallel & Distributed Processing Symposium (IPDPS), pp.1-8 (2010)

[26] Toshio Endo and Guanghao Jin: Software Technologies Coping with Memory Hierarchy of GPGPU Clusters for Stencil Computations. In Proceedings of IEEE Cluster Computing (CLUSTER2014), pp. 132–139, (2014).