Implementation and Evaluation of 2.5D Matrix Multiplication on K Computer

Daichi Mukunoki (daichi.mukunoki@riken.jp)
Toshiyuki Imamura (imamura.toshiyuki@riken.jp)
RIKEN Advanced Institute for Computational Science

(1) Introduction

Performance improvement of recent supercomputers relies on increasing the parallelism (i.e. the number of nodes or cores). On such highly parallel computers, the performance of a computation task could be communication-bound when the problem size per process is not large enough, and therefore communication avoiding techniques are required to improve the strong scaling performance. The 2.5D algorithm for parallel matrix multiplication (PDGEMM, C=αAB+βC) has been proposed [1] as one of such techniques. In this study, we have implemented a 2.5D parallel matrix multiplication using the SUMMA algorithm [2] and conducted the performance evaluation on the K computer (RIKEN AICS, JAPAN). A notable contribution of this study is that *our implementation is designed to perform the 2.5D algorithm on 2D distributed matrices on a 2D process grid*, and *it outperforms conventional 2D implementations (ScaLAPACK PDGEMM and 2D-SUMMA) even when the cost for matrix redistributions between 2D and 2.5D distributions is included*. Also, this study presents a detailed performance analysis of the 2.5D implementation by showing the breakdown of the execution time.

(2) 2.5D Matrix Multiplication

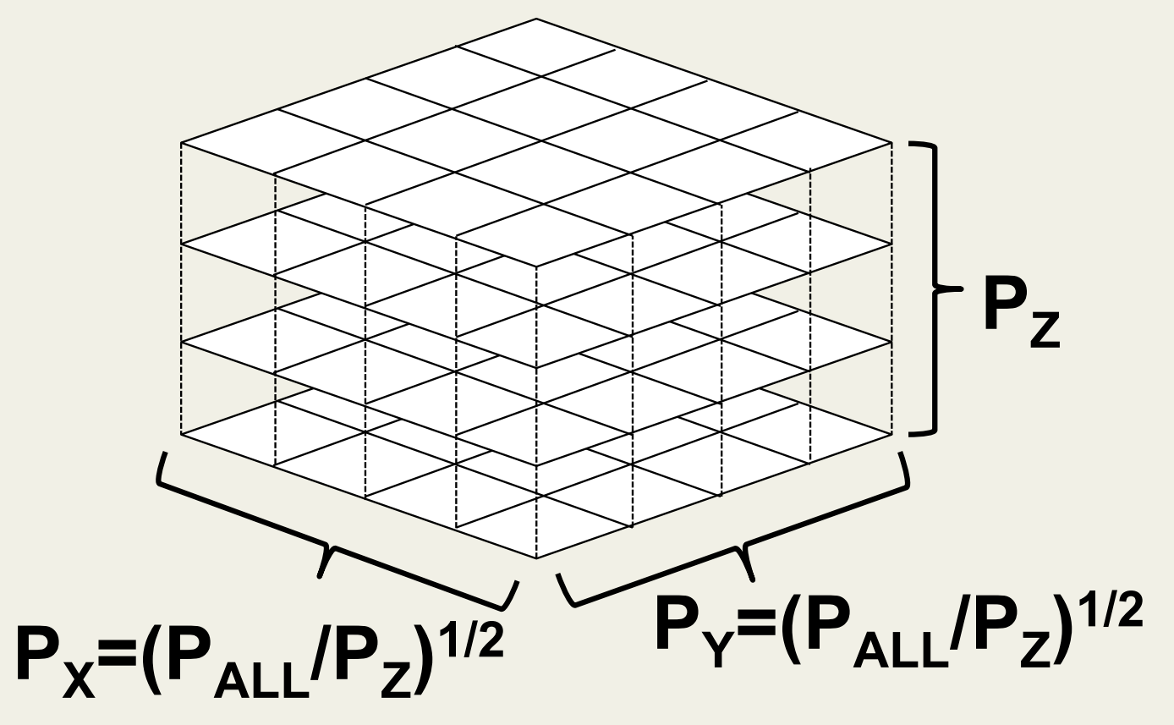


Figure 1. 3D process grid

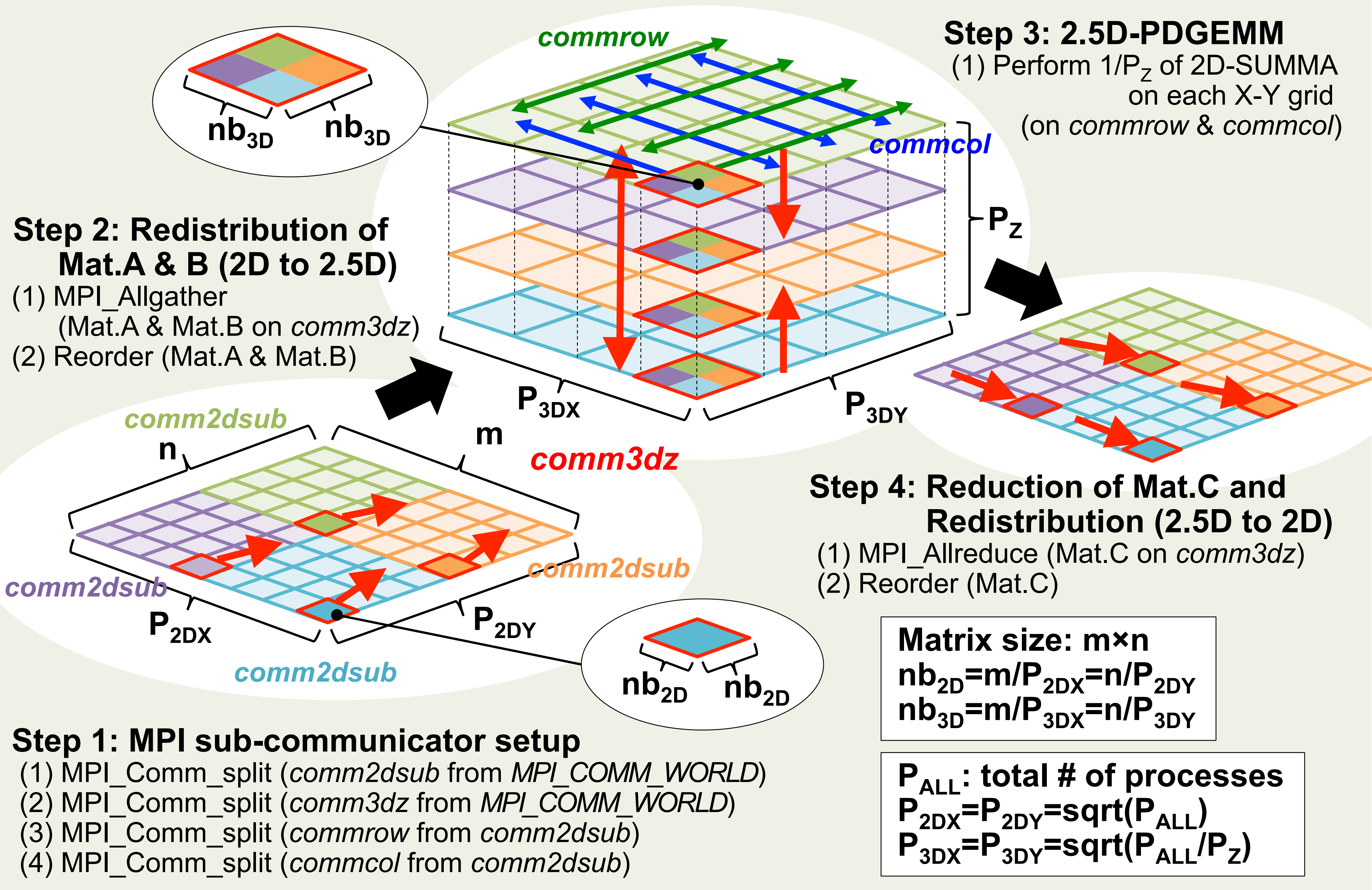
Table 1. Comparison of 2.5D with 2D algorithm

	2D	2.5D
Computation	$O(n^3/P_{ALL})$	$O(n^3/P_{ALL})$
Memory	$O(n^2/P_{ALL})$	$O(P_Z n^2/P_{ALL})$
Bandwidth	$O(n^2/P_{ALL}^{1/2})$	$O(n^2/(P_Z P_{ALL})^{1/2})$
Latency	$O(P_{ALL}^{1/2})$	$O((P_{ALL}/P_Z^3)^{1/2})$

- P_{ALL}: total number of processes
- P_Z: number of processes for z-dimension
- n: matrix size

The 2.5D algorithm uses a 3D process grid (P_X×P_Y×P_Z) as shown in **Figure 1** and stacks the matrices that distributed on a 2D (P_X-P_Y) process grid along the Z (vertical) direction: the matrices are duplicated P_Z-times on each P_X-P_Y grid. On each P_X-P_Y grid, 1/P_Z of a conventional parallel matrix multiplication algorithm is performed, and then, the final result is computed by reducing the temporal results on each P_X-P_Y grid among the P_Z processes. The details including the theoretical cost are described in the paper [1]. **Table 1** summarizes the theoretical costs of 2D and 2.5D algorithms.

(3) Implementation



Step 1: MPI sub-communicator setup
(1) MPI_Comm_split (*comm2dsub* from *MPI_COMM_WORLD*)
(2) MPI_Comm_split (*comm3dz* from *MPI_COMM_WORLD*)
(3) MPI_Comm_split (*commrow* from *comm2dsub*)
(4) MPI_Comm_split (*commcol* from *comm2dsub*)

Step 2: Redistribution of Mat.A & B (2D to 2.5D)
(1) MPI_Allgather (Mat.A & Mat.B on *comm3dz*)
(2) Reorder (Mat.A & Mat.B)

Step 3: 2.5D-PDGEMM
(1) Perform 1/P_Z of 2D-SUMMA on each X-Y grid (on *commrow* & *commcol*)

Step 4: Reduction of Mat.C and Redistribution (2.5D to 2D)
(1) MPI_Allreduce (Mat.C on *comm3dz*)
(2) Reorder (Mat.C)

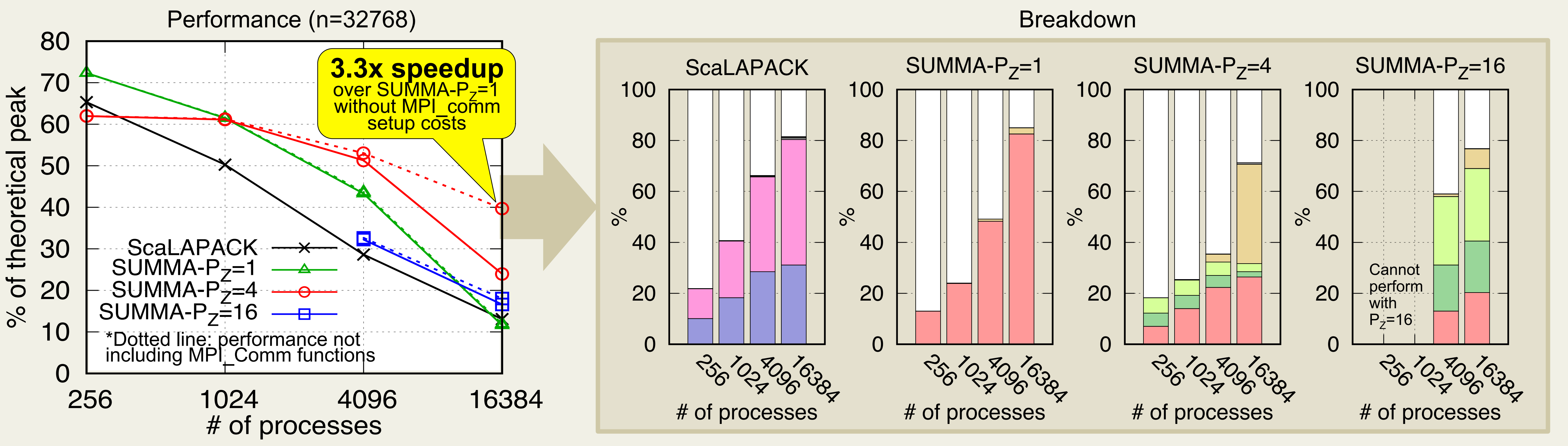
Matrix size: m×n
$nb_{2D}=m/P_{2DX}=n/P_{2DY}$
$nb_{3D}=m/P_{3DX}=n/P_{3DY}$

P_{ALL}: total # of processes
$P_{2DX}=P_{2DY}=sqrt(P_{ALL})$
$P_{3DX}=P_{3DY}=sqrt(P_{ALL}/P_Z)$

Figure 2. Overview of our 2.5D PDGEMM implementation

Our implementation is designed to perform the 2.5D algorithm on 2D distributed matrices on a 2D process grid. Therefore, it creates a 2.5D process grid from the 2D process grid and requires matrix redistributions between 2D and 2.5D distributions. **Figure 2** shows the overview of the implementation. The implementation consists of 4 steps. Step 1 is the MPI sub-communicator setup phase, and it is required only once in the case of calling the PDGEMM routine multiple times. Step 2 redistributes Matrices A & B from 2D to 2.5D by using MPI_Allgather. Step 3 performs 2.5D matrix multiplication. This study uses the SUMMA algorithm [2] as a parallel matrix multiplication algorithm. Finally, step 4 computes the final result of Matrix C by using MPI_Allreduce. Steps 2 and 4 require matrix reordering to fit the MPI collective communications. Note that our current implementation only supports square matrices, square process grid, and fixed nb size.

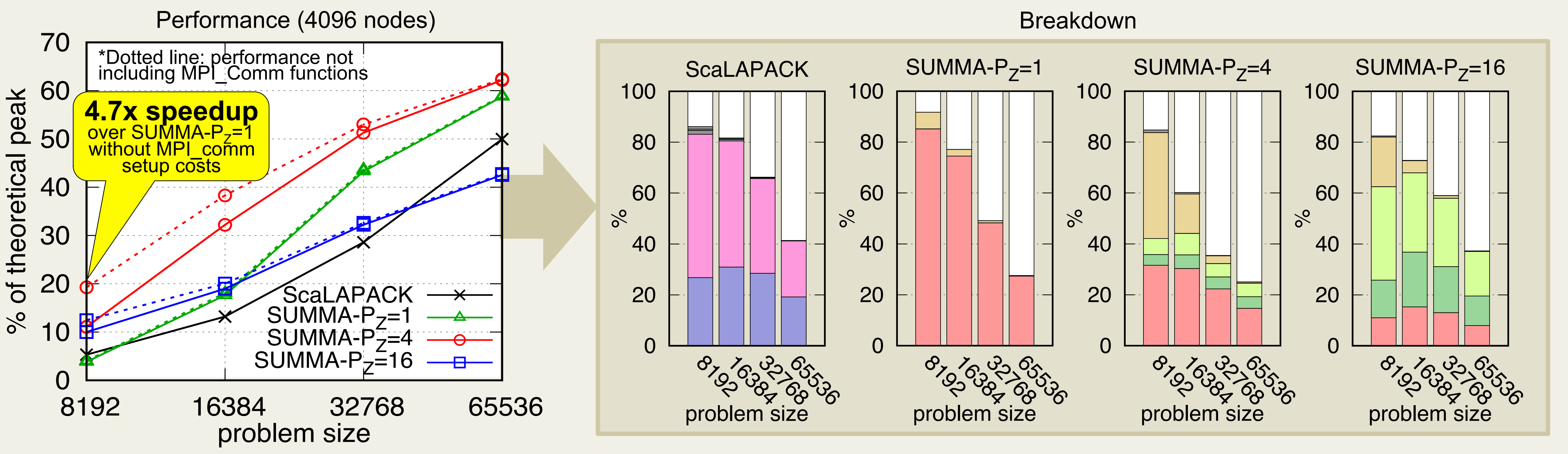
(4) Performance Evaluation



3.3x speedup over SUMMA-P_Z=1 without MPI_comm setup costs

*Dotted line: performance not including MPI_Comm functions

Figure 3. Performance comparison (strong scaling, n=32768)



*Dotted line: performance not including MPI_Comm functions

4.7x speedup over SUMMA-P_Z=1 without MPI_comm setup costs

Figure 4. Performance comparison (different problem sizes, 4096 nodes)

■ Evaluation environment and conditions

System	K computer (RIKEN AICS)
Environment version	K-1.2.0-21 (released: Jan. 10, 2017)
Processor (per node)	SPARC64 VIIIfx 8 cores, 2.0 GHz, 128 GFlops (double-precision)
Memory (per node)	DDR3 16GB, 64GB/s
Network	Tofu interconnect (6 dimensional mesh/torus) 5 GB/s for each direction
Compiler	mpiccpx
Compile options	-Xg -Kfast,parallel,openmp -O3 -MD -SCALAPACK –SSL2BLAMP
MPI & OpenMP configuration	1 MPI-process per node 8 threads per MPI-process (1 thread per core)
PJM configurations	#PJM -rsc-list “node=P_{2DX}xP_{2DY}”

- The performances are the best values obtained by executing a routine 3 times on a program and executing the program 2 times on a job script
- The values on the breakdown figures are average of the execution times on each thread obtained by Fujitsu’s Advanced Profiler (fapp)
- On ScaLAPACK PDGEMM, the block size: nb = n/P_{2DX}

■ Legend (breakdown of ScaLAPACK)

MPI_Send	MPI communication functions in ScaLAPACK PDGEMM
MPI_Recv	
MPI_Type_vector	Non-communication MPI functions
MPI_Type_commit	
MPI_Type_free	
Others	Others except for above the MPI functions (most of them are DGEMM cost)

■ Legend (breakdown of our SUMMA implementation)

MPI_Bcast	MPI communication function in SUMMA
MPI_Allgather	Redistribution from 2D to 2.5D
MPI_Allreduce	Reduction and Redistribution from 2.5D to 2D
MPI_Comm_split	MPI sub-communicator setup
MPI_Comm_size	Non-communication MPI functions
MPI_Comm_rank	
MPI_Comm_free	
Others	Others except for above the MPI functions (most of them are DGEMM cost)

- We conducted the performance evaluation on the K computer. **Figure 3** shows the strong scaling performance on n=32768 using 256 to 16384 nodes. **Figure 4** shows the performances of different problem sizes on 4096 nodes. Note that SUMMA-P_Z=1 corresponds the conventional 2D SUMMA implementation.
- SUMMA-P_Z=4 outperformed ScaLAPACK PDGEMM and 2D-SUMMA (P_Z=1): the 2.5D implementation is effective when the performance is communication bound even when including the cost for matrix redistributions between 2D and 2.5D distributions.
- The cost of MPI_Comm_split (required on the MPI sub-communicator setup phase: step 1 shown in Figure 2) is not negligible in the case of the problem size per process is small enough. This phase is required only once even when using the PDGEMM routine multiple times. Therefore, the phase should be provided as a separated function (an initialization function) or use of MPI sub-communicators should be avoided. The performances without MPI_Comm_*** functions are shown with dotted lines in the left side figures.
- On SUMMA-P_Z=16, the cost for the redistribution and reduction (steps 2 and 4 shown in Figure 2) increased and thus the performance degraded compared to the case of P_Z=4.

(5) Conclusion and Future Work

- The 2.5D implementation is effective to improve the strong scaling performance even when including the cost for matrix redistributions between 2D and 2.5D distributions when compared to the conventional 2D implementation.
- The cost of MPI_Comm_split is not negligible in the case of problem size per process is small enough.
- Future work includes overwrapping implementation [3], supporting for non-square process grid, auto-tuning for selecting implementations (2D or 2.5D) and the parameter P_Z (model-driven approach is also applicable), and performance evaluation on actual applications.

References:
[1] E. Solomonik and J. Demmel: Communication-Optimal Parallel 2.5D Matrix Multiplication and LU Factorization Algorithms, pp. 90–109 (2011).
[2] R. A. van de Geijn and J. Watts: SUMMA: Scalable Universal Matrix Multiplication Algorithm, Technical report, Austin, TX, USA (1995).
[3] E. Georganas, J. González-Domínguez, E. Solomonik, Y. Zheng, J. Touriño and K. Yelick: Communication Avoiding and Overlapping for Numerical Linear Algebra, Proc. International Conference on High Performance Computing, Networking, Storage and Analysis, SC'12, pp. 100:1–100:11 (2012).

Acknowledgement:
The results were obtained by using the K computer at the RIKEN Advanced Institute for Computational Science (project number: RA000022). This study is a part of the Flagship2020 project. We thank Akiyoshi Kuroda (RIKEN AICS), Eiji Yamanaka (Fujitsu Limited), and Naoki Sueyasu (Fujitsu Limited) for helpful suggestions and discussions.

ISC2017 June 18-21, Frankfurt, Germany