2.5D Matrix Multiplication

(2) 2.5D Matrix Multiplication

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 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 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.

Table 1. Comparison of 2.5D with 2D algorithm

<table>
<thead>
<tr>
<th></th>
<th>2D</th>
<th>2.5D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computation</td>
<td>O(n^3/P)</td>
<td>O(n^3/P^2)</td>
</tr>
<tr>
<td>Memory</td>
<td>O(n^2)</td>
<td>O(n^2)</td>
</tr>
<tr>
<td>Bandwidth</td>
<td>O(n/P)</td>
<td>O(n/P)</td>
</tr>
<tr>
<td>Latency</td>
<td>O(n/P^2)</td>
<td>O(n/P^2)</td>
</tr>
<tr>
<td>P_m = P(A)P(B)</td>
<td>P_m = P(A)P(B)</td>
<td></td>
</tr>
<tr>
<td>P_m = P(A)P(B)</td>
<td>P_m = P(A)P(B)</td>
<td></td>
</tr>
<tr>
<td>P_m = P(A)P(B)</td>
<td>P_m = P(A)P(B)</td>
<td></td>
</tr>
</tbody>
</table>

(3) Implementation

Our implementation is designed to perform the 2.5D algorithm on 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_Allgatherv. Step 3 performs 2.5D matrix multiplication. This study uses the SUMMA algorithm [2] as a parallel matrix multiplication algorithm.

Figure 2. Overview of our 2.5D PDGEMM implementation

(4) Performance Evaluation

We conducted the performance evaluation on the K computer. Figure 3 shows the strong scaling performance on n=32768 on 256 to 16384 nodes. Figure 4 shows the performances of different problem sizes on 4096 nodes. Table 1 includes the theoretical cost for selecting implementations (2D or 2.5D) and the performances of different problem sizes on 4096 nodes. Note that SUMMA-P_z=1 corresponds the conventional 2D SUMMA implementation.

The cost of MPI_Comm_split (required on the MPI sub-communicator setup phase: step 1 shown in Figure 2) is negligible in the case of the problem size per process is small enough. This phase is required only once 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***_split 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.

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

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

(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 overramping 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:

Acknowledgement:
The results were obtained using the K computer at the RIKEN Advanced Institute for Computational Science (project number: K40002). This study is a part of the Flagship2020 project. We thank Akiyuki Komura (RIKEN AICS), Eiji Yamada (Fujitsu Limited), and Naoki Sueyasu (Fujitsu Limited) for helpful suggestions and discussions.

(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 communication bound computations is significantly improved by reducing the communication size per process that 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 performs conventional 2D implementations (ScaLAPACK 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.