Analyzing Performance of BiCGStab with Hierarchical Matrix on GPU clusters

Ichitaro Yamazaki*, Ahmad Abdelfattah*, Akihiro Ida†, Satoshi Ohshima‡,
Stanimire Tomov*, Rio Yokota§, and Jack Dongarra*
*University of Tennessee, Innovative Computing Laboratory, USA
†University of Tokyo, Information Technology Center, Japan
‡Kyushu University, Research Institute for Information Technology, Japan
§Tokyo Institute of Technology, Global Scientific Information and Computing Center Japan

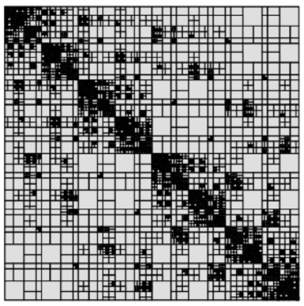
***Abstract*—ppohBEM is an open-source software package implementing the boundary element method. One of its main software tasks is the solution of the dense linear system of equations, for which, ppohBEM relies on another software package called HACApK. To reduce the cost of solving the linear system, HACApK hierarchically compresses the coefficient matrix using adaptive cross approximation. This hierarchical compression greatly reduces the storage and time complexities of the solver and enables the solution of large-scale boundary value problems. To extend the capability of ppohBEM, in this paper, we carefully port the HACApK's linear solver onto GPU clusters. Though the potential of the GPUs has been widely accepted in high-performance computing, it is still a challenge to utilize the GPUs for a solver, like HACApK's, that requires fine-grained computation and global communication. First, to utilize the GPUs, we integrate the batched GPU kernel that was recently released in the MAGMA software package. We discuss several techniques to improve the performance of the batched kernel. We then study various techniques to address the inter-GPU communication and study their effects on state-of-the-art GPU clusters. We believe that the techniques studied in this paper are of interest to a wide range of software packages running on GPUs, especially with the increasingly complex node architectures and the growing costs of the communication. We also hope that our efforts to integrate the GPU kernel or to setup the inter-GPU communication will influence the design of the future-generation batched kernels or the communication layer within a software stack.**

I. Introduction

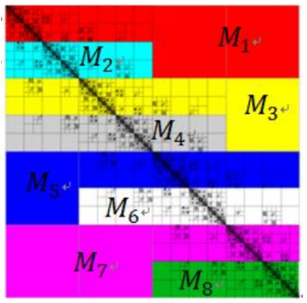
The boundary integral equations is a powerful tool for solving the boundary value problems of partial differential equations. It has been successfully used in many scientific and engineering applications including in the studies of acoustics, electromagnetics, fracture mechanics, and fluid mechanics. ppohBEM [1] is an open-source software package that implements the boundary element method (BEM) for numerically solving the integral equations. For the numerical solution, ppohBEM relies on the HACApK library [2], whose main purpose is the solution of the dense linear system of equations. If the dense linear system had been directly solved, the size of the problems that ppohBEM can solve would have been limited by the excessive amount of the required memory and time. To reduce the costs of solving the linear system, HACApK takes advantage of the special properties of the problem and hierarchically compresses the coefficient matrix using low-rank factorization of its off-diagonal blocks. This hierarchical compression greatly reduces the storage and time complexities of the solver, and increases the size of the problems that ppohBEM can solve.

ppohBEM has been mainly designed for the homogeneous distributed-memory computers. In order to extend the capability of HACApK, in this paper, we port the HACApK's linear solver onto a GPU cluster. Though the potential of the GPUs has been widely accepted in high-performance computing, it is not trivial to utilize the GPU cluster for a solver, like HACApK's, that requires fine-grained computation and global communication. Due to the tremendous compute power available on each node, the data communication between the GPUs or any part of the solver that is not executed or does not perform well on the GPU, can quickly become a performance bottleneck. To effectively make use of the GPUs, in this paper, we carefully design our implementation.

To harness the compute power of each node, we first integrate a batched GPU kernel into HACApK and utilize the multiple GPUs available on the node for executing many small-size independent tasks in parallel. We discuss several techniques to improve the performance of the GPU kernel: e.g., 1) sorting the computational tasks to reduce the overhead associated with the variable shapes of the tasks, 2) dividing the tasks into multiple batches with different task counts, and 3) using GPU streams to execute multiple batches in parallel. We then evaluate several techniques to address the inter-GPU communication: 1) different schemes to assign each process to multiple GPUs, 2) usage of NVLink for the data exchanges among the local GPUs, and 3) overlapping the communication with computation. Our experimental results on two state-of-the-art GPU clusters demonstrate the effects of these techniques. We believe that the techniques studied in this paper (to utilize the multiple GPUs on the node and to manage the communication between the GPUs) are of interest to a wide range of applications running on GPUs, especially with the increasingly complex node architectures and the growing cost of the communication compared with the computation. Furthermore, we intentionally rely on existing open-source software packages, instead of developing specialized kernels,



(a) Matrix partition for `100ts`.



(b) Matrix distribution example.

Fig. 1. $\mathcal{H}$-matrix partition and distribution; j-th process owns submatrix M_j.

to show their current potential and limitations. We hope that our efforts to integrate the GPU kernel or to setup the inter-GPU communication will influence the design of the future-generation batched kernels or the communication layer within the software stack.

The rest of the paper is organized as follows. We first introduce, in Sections II and III, ppohBEM and HACApK's linear solver, respectively. We then, in Section V, integrate the GPU kernel into HACApK, and address the inter-GPU communications in Section VI. Finally, in Section VII, we discuss the challenges of avoiding or hiding the communication in HACApK. Our final remarks are listed in Section VIII.

II. ppohBEM and HACApK

ppohBEM [1] implements BEM for solving boundary integral equations. For the numerical solution of the integral equations, ppohBEM uses HACApK [2] for solving a linear system of equations, $A\mathbf{x} = \mathbf{b}$, whose coefficient matrix A is dense. The sizes of the problems that ppohBEM can solve is limited by the excessive amount of the memory and time needed for solving the dense linear system. To reduce the costs of solving the linear system, HACApK exploits the property that the kernel function $g(\mathbf{p}_1, \mathbf{p}_2)$ of the integral operator for two far-away points $\mathbf{p}_1$ and $\mathbf{p}_2$ can be approximated by a degenerate kernel. Hence, many of the off-diagonal blocks of the coefficient matrix can be well approximated by its low-rank factorization. To extract this hierarchal low-rank structure of the matrix A, HACApK first uses the geometric information of the problem to generate a suitable matrix permutation and matrix partition based on a cluster tree such that off-diagonal blocks of large dimensions become low-rank. Then, the low-rank representation of each block is computed by algebraically approximating the quadrature of the kernel using Adaptive Cross Approximation (ACA) or ACA+. Figure 1(a) shows the partition of one of our test matrices. This hierarchical low-rank compression of the matrix reduces both the storage and computational costs associated with the matrix (e.g., matrix-vector multiply with $O(n \log(n))$ storage and flops where n is the number of unknowns in the matrix). The hierarchically-compressed matrix is commonly referred to as $\mathcal{H}$-matrix.

III. BiCGStab

```
 1: $\alpha := 0.0$; $\beta := 0.0$; $\zeta := 0.0$;
 2: $\mathbf{t} := A\mathbf{x}$
 3: $\mathbf{r} := \mathbf{b} - \mathbf{t}$; $\mathbf{r}_0 := \mathbf{r}$
 4: $\gamma := \|\mathbf{r}_0\|_2$
 5: for $iter = 1, 2, \ldots, maxiters$ do
 6:     if $\gamma/\|\mathbf{r}_0\|_2 < tol$ then
 7:         break;
 8:     end if
 9:     $\mathbf{p} := \mathbf{r} + \beta \cdot (\mathbf{p} - \zeta \cdot \mathbf{v})$
10:     $\mathbf{v} := A\mathbf{p}$, followed by Allgatherv
11:     $\alpha = (\mathbf{r}_0, \mathbf{r})/(\mathbf{r}_0, \mathbf{v})$
12:     $\mathbf{v} := \mathbf{r} - \alpha \cdot \mathbf{v}$
13:     $\mathbf{t} := A\mathbf{v}$, followed by Allgatherv
14:     $\zeta = (\mathbf{t}, \mathbf{v})/(\mathbf{t}, \mathbf{t})$
15:     $\mathbf{x} := \mathbf{x} + \alpha\mathbf{p} + \zeta\mathbf{v}$
16:     $\mathbf{r} := \mathbf{v} - \zeta\mathbf{t}$
17:     $\beta = \alpha/\zeta \cdot (\mathbf{r}_0, \mathbf{r})/\gamma$
18:     $\gamma = \|\mathbf{r}\|$
19: end for
```

(a) Standard.

```
 1: $\beta := 0.0$; $\zeta := 0.0$;
 2: $\mathbf{t} := A\mathbf{x}$
 3: $\mathbf{r} := \mathbf{b} - \mathbf{t}$; $\mathbf{r}_0 := \mathbf{r}$
 4: $\mathbf{f} := A\mathbf{r}_0$, $\mathbf{t} := A\mathbf{f}$
 5: $\alpha = (\mathbf{r}, \mathbf{r})/(\mathbf{r}, \mathbf{f})$
 6: $\gamma := \|\mathbf{r}_0\|_2$
 7: for $iter = 1, 2, \ldots, maxiters$ do
 8:     if $\gamma/\|\mathbf{r}_0\|_2 < tol$ then
 9:         break;
10:     end if
11:     $\mathbf{p} := \mathbf{r} + \beta \cdot (\mathbf{p} - \zeta \cdot \mathbf{g})$
12:     $\mathbf{g} := \mathbf{f} + \beta \cdot (\mathbf{g} - \zeta \cdot \mathbf{z})$
13:     $\mathbf{z} := \mathbf{t} + \beta \cdot (\mathbf{z} - \zeta \cdot \mathbf{v})$
14:     $\mathbf{q} := \mathbf{r} - \alpha \cdot \mathbf{g}$
15:     $\mathbf{y} := \mathbf{f} - \alpha \cdot \mathbf{z}$
16:     $\mathbf{v} := A\mathbf{z}$, hide Allgatherv behind $\zeta$
17:     $\zeta := (\mathbf{q}_i, \mathbf{y}_i)/(\mathbf{y}_i, \mathbf{y}_i)$
18:     $\mathbf{x} := \mathbf{x} + \alpha\mathbf{p} + \zeta\mathbf{q}$
19:     $\mathbf{r} := \mathbf{q} - \zeta \cdot \mathbf{y}$
20:     $\mathbf{f} := \mathbf{y} - \zeta \cdot (\mathbf{t} - \alpha\mathbf{v})$
21:     $\mathbf{t} := A\mathbf{v}$, hide Allgatherv with $\alpha, \beta, \gamma$
22:     $\beta = \alpha/\zeta \cdot (\mathbf{r}_0, \mathbf{r})/(\mathbf{r}_0, \mathbf{r})$
23:     $\alpha = (\mathbf{r}_0, \mathbf{r})/((\mathbf{r}_0, \mathbf{f})$
            $+\beta \cdot ((\mathbf{r}_0, \mathbf{g}) - \zeta(\mathbf{r}_0, \mathbf{z}))$
24:     $\gamma = \|\mathbf{r}\|$
25: end for
```

(b) Pipelined.

Fig. 2. Pseudocode of BICGStab.

To solve the linear system, HACApK relies on the BiConjugate Gradient Stabilized (BiCGStab) [3], a Krylov subspace projection method for solving a general linear system of equations (see Figure 2(a) for its pseudocode). Though the low-rank compression reduces the cost of the matrix multiply, in many cases, the BiCG's iteration time is still dominated by this Hierarchical Matrix Vector multiply (`HiMV`). To reduce the iteration time using a distributed-memory computer, HACApK distributes the contiguous, but not disjoint, rows of the matrix among the processes (see Figure 1(b) for an illustration). Then, each process performs `HiMV` with its local submatrix. All other vector operations of the BiCG iteration are redundantly performed by all the processes. With this parallelization scheme, the only required inter-process communication is the all-gather needed after `HiMV` to form the global vector on each process (using `MPI_Allgatherv`). This parallelization scheme is motivated by two performance properties of the solver: 1) the BiCG's computation time is dominated by `HiMV`, while the time needed for the remaining vector operations is insignificant in the computation time and 2) the redundant computation of the vector operations avoids the global all-reduces needed to compute the six dot-products for each BICG iteration, which can be much more expensive compared with the arithmetic operations. Hence, by redundantly performing the vector operations, this parallelization scheme aims to balance out two conflicting performance factors: distributing the computation with a minimum inter-process communication.

HACApK also supports a hybrid MPI/OpenMP mode. During the BiCG iteration, OpenMP threads are used to parallelize the vector operations and to parallelize `HiMV` by executing the small independent matrix-vector multiplies with the local dense and compressed blocks in parallel. This not only reduces the serial bottleneck of performing the vector operations but also improves the parallel scalability by reducing the process

name	Sphere objects 100ts	288ts	338ts	1ms
size, n	101,250	288,000	338,000	1,004,400
compress%	16.0	16.7	16.9	17.6

	Sphere objects (precond)		Human objects		
name	8ms	20ms	hum2	hum4	hum6
size, n	7,996,800	20,736,000	78,656	314,624	707,904
compress%	1.7	1.7	17.3	22.9	31.7

Fig. 3. Test matrices where "compression%" is the ratio of the total number of numerical values in the compressed matrix over $n \log_2(n)$.

count and lowering the inter-process communication cost. In order to extend the capability of ppohBEM, in this paper, we port the HACApK's BiCG solver onto GPU clusters.

IV. Experiment Setups

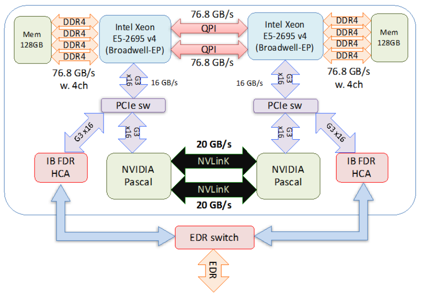
We conducted all the experiments in double precision, and used the matrices from electrostatic field simulations with perfect conductors of two particular shapes:

- **Sphere**: pairs of perfect conductors with the shape of a sphere. For each pair, one sphere has its electric potential set to be 1 Volt, while the other has the electric potential of -1 Volt. We use the boundary value of 0 Volt at infinity and analyze the induced electrical charge on the surface of the spheres.
- **Human**: perfect conductors with the shape of a humanoid who is standing on a uniform 2D grid on a uniform electric field. The surface of the humanoid is divided into $2,359,680$ triangular elements and the induced electrical charge on the humanoid's surface was calculated using an indirect BEM with a single layer potential formulation and step functions as the base function for the BEM.

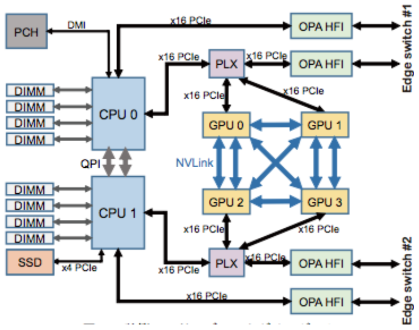
Figure 3 lists our test matrices. The large-scale matrices `8ms` and `20ms` were used for the inner-iteration to precondition the linear system [4]. All of their compressed blocks have rank one and we fixed the number of inner iterations to be 20 for our experiments. We denote the k-th block of the matrix using $B^{(k)}$ and set the threshold for ACA+ such that $B^{(k)}$ is approximated to $\|B^{(k)} - U^{(k)}V^{(k)}\| \leq 10^{-3}$. The computed solution is considered to have converged when the residual ℓ_2-norm is reduced by at least seven order of magnitude. These are the standard matrices used in the previous studies, and they are the typical setups used in the actual simulation.

We conducted all of our experiments on either the Reedbush-H supercomputer at the University of Tokyo or the Tsubame-3 supercomputer at Tokyo Institute of Technology. Each node of both supercomputers has NVIDIA Telsa P100 GPUs, each of which has the double-precision peak performance of 4.7 Tflop/s and 16 GB of main memory with the peak bandwidth of 732 GB/s. We obtained 4.7 Tflop/s using `cublasDgemm` for a large enough matrix and the bandwidth of 495 GB/s using NVIDIA's bandwidth utility.

- Each node of Reedbush-H consists of two eighteen-core Intel Xeon E5-2695 V4 (Broadwell-EP) processor and two P100 GPUs. These two GPUs on the node are connected by two NVLinks with the theoretical peak bandwidth of 2×20 GB/s, while the data copy between the CPU and the GPU goes through the PCI Express (PCIe) Gen3 with the bandwidth 16 GB/s (and observed bandwidth of 11.1 or 12.9 GB/s for copying data to or from the GPU, respectively). The nodes are connected by the InfiniBand FDR with the bandwidth of 2×56 Gb/s.
- Each node of Tsubame-3 has two fourteen-core Intel Xeon E5-2680 V4 (Broadwell-EP) processors and four NVIDIA P100 GPUs. The CPU and the GPUs are connected through the PCIe's where a pair of the GPUs share the same PCIe. The four GPUs are also directly connected through an NVLink except for two pairs of the GPUs that are connected by two NVLinks. The nodes are connected by the Intel Omni-Path with the bandwidth of 4×100 Gb/s.



(a) Reedbush-H.



(b) Tsubame 3.

Fig. 4. Node architectures of our testbeds.

Figure 4, taken from [5], [6], illustrates the node architectures of these two systems.

On both systems, we complied our code using OpenMPI `mpicc` version 2.1.1 compiler with Intel `icc` version 17.0 and CUDA `nvcc` version 8.0.44 compilers. The optimization flags `-O3` was used with the additional flag `-xCORE-AVX2` on Reedbush-H. Though there have been significant efforts to utilize both the CPU and GPUs, in this work, we decided to perform all the computation on the GPUs (while using the CPU for scheduling the computational and communication tasks). This is motivated by our observations that on many leadership supercomputers, the gaps between the compute powers of the CPUs and GPUs on the node are widening. For example, on the early-access version of the Summit supercomputer (i.e., Summitdev) at Oak Ridge Leadership Computing Facility, each node has four P100 GPUs and two ten-core POWER8 CPUs. The difference in their double-precision peak performances is $33\times$ (i.e., 4×4.7 Tflop/s on the GPUs vs. 560 Gflop/s on the CPUs). In the full production system, the gap likely grows having six NVIDIA Volta GPUs and two ten-core POWER9 CPUs on each node. On such architectures, any operation on the CPUs could quickly become a performance bottleneck.

V. Integrating Batched GPU Kernel

The BiCG's computation time is typically dominated by `HiMV` that consists of many small matrix-vector multiplies with the dense or compressed blocks of the matrix. To accelerate `HiMV`, we use the batched GPU kernel of MAGMA that performs a batch of small dense matrix-vector multiplies (`dgemv`'s) through one kernel launch. We describe our implementation of the batched kernel in Section V-A and present how we integrated the kernel into HACApK in Section V-B. We then, in Section V-C, show the effect of the GPU kernel on the BiCG's performance. Here, we focus on improving the performance of each process using a GPU (i.e., one GPU per process), while in Section VI, we look at improving the performance on the GPU cluster.

A. Variable-size Batched dgemv Kernel for GPU

The variable size batched `dgemv` kernel, or simply `dgemv_vbatched`, was part of the MAGMA 2.1 release. Figure 5 shows the overall structure of `dgemv_vbatched`, which can be used for any other kernel. The design of the batched kernel has three components. The first is the *computational code*, which carries out a single `dgemv` operation. This code can be used with either batched or non-batched kernel, and is written as a CUDA device routine. We use C++ templates to have a highly tunable code that achieves a nearly optimal performance for all input sizes, which has been already demonstrated for fixed size batch workloads [7]. The second component is the *Adaptive SubGrid Truncation (ASGT)*, which has been first introduced for level-3 BLAS kernels [8]. The ASGT technique is an upper layer that encloses the computational code. As shown in Figure 5, it is a distribution layer that organizes the kernel into a number of subgrids, and assigns one `dgemv` operation per subgrid. It is also a protection layer that prevents the computational code from committing out-of-bound memory accesses. It does so by truncating each subgrid to match the size of the assigned problem. Because `dgemv_vbatched` must accommodate the largest matrix in the batch, other subgrids that are assigned to smaller problems might possess thread blocks without any work. The ASGT layer ensures that such thread blocks are detected and terminated before starting the computational code.

The final component is the kernel driver. The driver must have the size of the largest matrix in the batch, which is necessary for the kernel configuration. For `dgemv_vbatched`, only the maximum number of rows is required. The driver

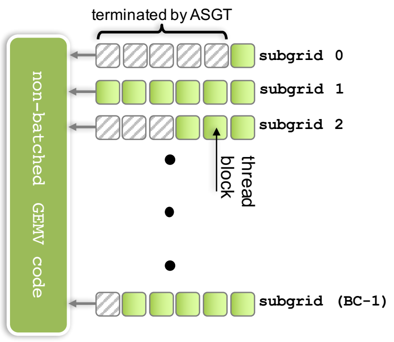


Fig. 5. Structure of the variable size `dgemv` kernel with batch count `BC`.

also has an error checking mechanism that is similar to BLAS. However, an important design aspect of the `dgemv_vbatched` routine is that both finding the maximum size(s) and error checking are decoupled from the bottom two layers, and can be skipped in order to achieve a better performance. In our case, we provide the maximum number of rows explicitly to the low-level API. We also skip the error checking phase, which is of a significant overhead, especially with a large batch count.

Figure 5 shows that there is a number of thread blocks that are launched, but then get immediately terminated using the ASGT technique. The overhead of having and terminating quickly these thread blocks is proportional to the variations in the number of rows across the matrices in the batch. While the overhead can be of less impact if the kernel is compute bound [9], it can become quite significant for memory bound kernels, such as the `dgemv` kernel that we use in this work. We pay attention to this effect while calling `dgemv_vbatched` from HACApK solver.

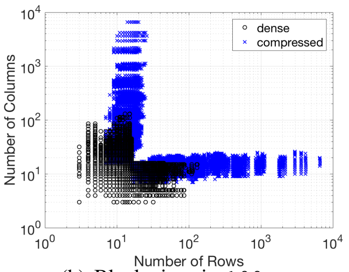
B. Integration into HACApK

We now describe how we integrated `dgemv_vbatched` into HACApK. In this section, we let each MPI process use one GPU, while in Section VI, we look at having multiple GPUs per process. To avoid the expensive data copy between the CPU and GPU, once the right-hand-side vector is copied to the GPU, all the matrix and vector operations are performed on the GPU. Our focus is on `HiMV`, which dominates the compute time of the iteration, while the rest of the vector operations are redundantly performed by all the processes. The only potential data copies between the CPU and GPU are for the inter-process communication to form the global vector on each GPU after the distributed `HiMV` (addressed in the next section).

Figure 6(a) shows the pseudocode of `HiMV`, while Figure 6(b) shows the sizes of the blocks (either dense $B^{(k)}$, or compressed $U^{(k)}$ and $V^{(k)}$) in the matrix `100ts`. We see a wide range of the block sizes where all the blocks on the diagonal are square and dense, while off-diagonal blocks can be either dense or compressed and are either tall-skinny or wide-short. To utilize the GPU, each process divides its local `dgemv` tasks with $B^{(k)}$, $U^{(k)}$, and $V^{(k)}$ into several

```
1: for $k = 1, 2, \ldots, n_\ell$ do
2:     if dense block then
3:         $\mathbf{y}^{(k)} := B^{(k)}\mathbf{x}^{(k)}$
4:     else
5:         $\mathbf{t}^{(k)} := V^{(k)}\mathbf{x}^{(k)}$
6:         $\mathbf{y}^{(k)} := U^{(k)}\mathbf{t}^{(k)}$
7:     end if
8: end for
```

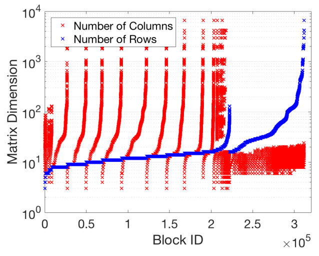
(a) HiMV pseudocode.



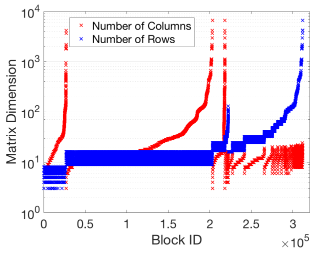
(b) Block sizes in 100ts.

Fig. 6. Matrix-vector multiply $\mathbf{y} := A\mathbf{x}$ with a $\mathcal{H}$-matrix A, where n_ℓ is the number of blocks in A, and $B^{(k)}$ is the k-th block with the corresponding vectors $\mathbf{x}^{(k)}$ and $\mathbf{y}^{(k)}$, and $U^{(k)}V^{(k)}$ is the low-rank representation of the k-th block, i.e., $B^{(k)} \approx U^{(k)}V^{(k)}$.

batches (e.g., a batch with a fixed number of dgemv's), and then calls dgemv_vbatched for each batch. To this end, we must resolve two types of data conflicts. First, the output vectors $\mathbf{y}^{(k)}$ of different dgemv's may overlap on each other. These data conflicts were resolved using the CUDA's atomic-add operation on the output vector $\mathbf{y}$. We found that on the latest NVIDIA GPU, the overhead of the atomic operation is minimum. The second type of the data conflicts is that when multiplying with the compressed block, the second dgemv with $U^{(k)}$ depends on the output vector $\mathbf{t}^{(k)}$ from the first dgemv with $V^{(k)}$. In order to resolve this conflict, we create two types of the batches; the first type only contains the dgemv's with the dense blocks $B^{(k)}$ and with the first low-rank block $V^{(k)}$. Then, once all the batches of the first type are launched, we launch the second type that contains the dgemv's with $U^{(k)}$. The data conflicts are resolved either by launching all the kernels on the same GPU stream or by using CUDA events to follow the data dependency on the multiple streams.



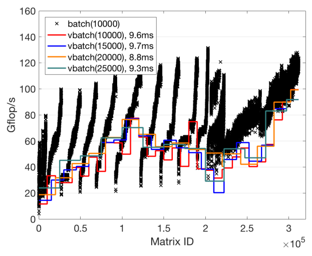
(a) sorted by number of rows.



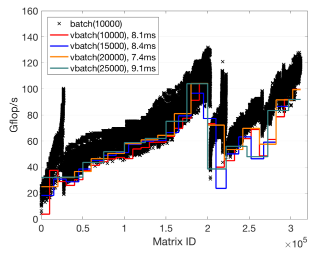
(b) grouped and sorted by number of rows, and then by number of columns within group.

Fig. 7. Different schemes to sort block sizes for the matrix 100ts.

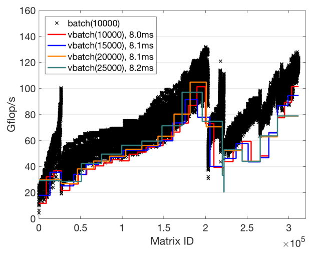
As discussed in Section V-A, dgemv_vbatched can execute dgemv's with variable matrix sizes in a single kernel launch. However, the performance of dgemv_vbatched can be much lower than its fixed-size counterpart. This is especially true when there is a wide range of matrix sizes in the single batch. In order to improve the performance of dgemv_vbatched, we examined several schemes to sort the blocks of A, on which dgemv's operate. Figure 7 shows



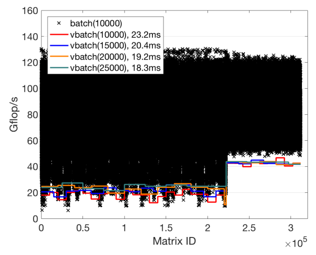
(a) sorting scheme of Figure 7(a) with fixed batch count.



(b) sorting scheme of Figure 7(b) with fixed batch count.



(c) sorting scheme of Figure 7(b) with variable batch counts.



(d) original without sorting.

Fig. 8. Performance of fixed-size and variable-size batched kernel, batch(ℓ) and vbatch(ℓ), on an NVIDIA P100 GPU, where ℓ is the batch count. In the legend, we show the total time required to execute all the tasks. We lauched batch(ℓ) for each block size whose performance is considered as the upper bound on the performance of vbatch(ℓ).

two promising sorting schemes: in Figure 7(a), the blocks were sorted in the ascending order of their numbers of rows, and in Figure 7(b), we first grouped the blocks according to the number of rows (the k-th group contains the block with the number of rows in the range between $8(k-1)+1$ and $8k$), and then we order the blocks in the same group according to their numbers of columns. Figures 8(a) and 8(b) show the effects of these two sorting schemes on the kernel performance for the matrix 100ts, while Figure 8(d) shows the original performance without sorting. The figure also shows the performance of the fixed-size batched kernel for each block size, which we consider as the upper bound on the performance of the variable-size kernel. We clearly see that the performance can be significantly improved by properly sorting the blocks (speedups of up to 2.5×) and the variable-size kernel may obtain the performance closer to that of the fixed-size kernel.

In the experiments so far, we used a fixed batch count (i.e., each batch contains the same number of tasks). Unfortunately, even after sorting the blocks, there may be a wide range of block sizes within a batch. For example, in Figure 8, the kernel spends longer time on the right side of the plot where the blocks are larger with wider variations in the numbers of rows (see Figure 7). To avoid the overhead associated with the different block sizes, in Figure 8(c), we adjust the batch counts such that each batch only contains a specific range of block sizes (i.e., we used three ranges $[1, 7]$, $[8, 31]$, and $[32, m^{(k)}_{\max}]$, which match the internal optimization of dgemv_vbatched, where $m^{(k)}_{\max}$ is the largest number of rows in the blocks).

	100ts	338s	hum2	hum6
sequential MKL + 32 OpenMP threads	10.42	11.52	11.51	10.68
CUBLAS + 5 streams	0.42	0.52	0.37	0.60
batch(5K, fixed) with pad	53.00	64.51	43.11	72.13
vbatch(20K, fixed)	65.21	74.10	50.93	84.35
vbatch(20K, variable) + 1 stream	51.43	56.67	40.72	70.50
vbatch(20K, variable) + 2 streams	81.12	84.14	72.20	96.56
vbatch(20K, variable) + 3 streams	84.13	86.14	72.20	96.87

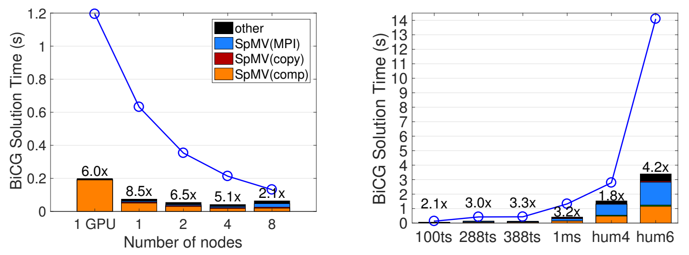
Fig. 9. Performance (Gflop/s) of different implementations with one NVIDIA P100 GPU and two 16-core Intel Broadwell CPUs on Reedbush-H.

Though the variable batch counts did not significantly improve the performance in this particular setup, we further explore its potential in the next paragraph.

Table 9 shows the overall performance of different implementations of HiMV. Our batched kernel (using the sorting scheme of Figure 8(b)) obtains much higher performance compared with that obtained by launching dgemv of the sequential MKL from OpenMP parallel for-loop or by launching cublasDgemv with multiple GPU streams. Even for a batch with the largest blocks, the batched kernel was faster than the other implementations. We also tested the fixed-size batched kernel such that all the tasks have the maximum matrix dimension in each batch. Though this avoids the overhead of the variable-sized tasks, the performance was lower due to the wasted operations. The last three rows of the table show the performance with the variable batch counts. Here, we split the blocks into smaller batches (i.e., the number of rows are split into the ranges of $8, 32, 64, 96, \dots, m^{(k)}_{\max}$). Then, to effectively utilize the GPU, we execute these small batches in parallel using multiple GPU streams. Compared with the fixed batch count, this variable batch counts now obtained higher performance. We note that the dgemv's performance is limited by the bandwidth where our GPU has the observed bandwidth of 495 GB/s. Since dgemv reads one matrix entry $a_{i,j}$ and two vector entries x_j and y_i to perform two flops, $y_j = a_{i,j} \cdot x_j + y_j$, the peak dgemv performance is between $61 \sim 183$ Gflop/s depending on if the vector entries stayed in the cache. We observed that cublasDgemv obtains around 150 Gflop/s for a large enough matrix. dgemv_vbatched obtained about $48 \sim 65\%$ of the peak cublasDgemv performance.

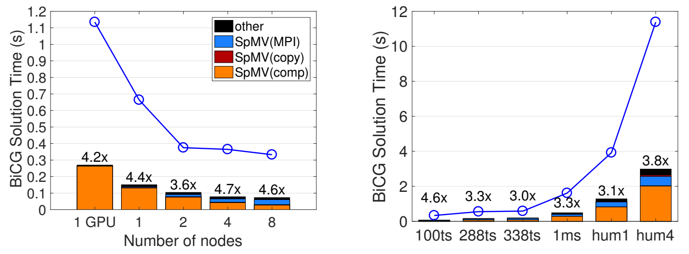
C. BiCG Performance

Figures 10 and 11 show the effects of the GPU kernels on the BiCG performance on Tsubame-3 and Reedbush-H, respectively. For the performance without the GPUs, we bind each process to a socket and launch one OpenMP thread on each of the available cores of the socket. We found this process/thread configuration typically gives the best performance of the hybrid MPI/OpenMP implementation. With the GPUs, we launch one process per GPU (i.e., four or two processes per node on Tsubame-3 or Reedbush-H). The figures clearly show that the GPUs have reduced the iteration time significantly, obtaining the speedups of about $4.2\times$ and $4.5\times$ on eight nodes of Tsubame-3 and Reedbush-H, respectively (in Figures 10(b) and 11(b)). At the same time, even on these small numbers of nodes, the communication starts to become significant, spending over 46% and 43% of the iteration time



(a) Strong scaling with 100ts. (b) Performance on 8 nodes.

Fig. 10. Performance on Tsubame-3. The blue markers show the solution time with original HACApK without GPUs, while the bars are with the GPUs.



(a) Strong scaling with 100ts. (b) Performance on 8 nodes.

Fig. 11. Performance on Reedbush-H. The blue markers show the solution time with original HACApK without GPUs, while the bars are with the GPUs.

on Tsubame-3 and Reedbush-H, respectively (e.g., on 1 node with the matrix 100ts, we obtained a speedup of about $8.5\times$ using GPUs on Tsubame-3). Furthermore, the part of the bar, colored in black, is mostly for the vector operations. Since these operations are not parallelized, they could become significant in the iteration time on a large number of GPUs.

VI. Improving Performance on Multiple GPUs

As the GPU kernel reduces the compute time on the node, the inter-process communication becomes more significant in the iteration time. This is especially true with HiMV because the hierarchical compression reduces the computational complexity, but the communication cost stays the same (i.e., $O(n \log(n))$ flops, compared with $O(n)$ communication volume since HACApK forces that the ranks of all the compressed block are at lest one). To reduce the communication cost, in this section, we develop a "multi-GPU" implementation of the solver where each process manages multiple GPUs. This multi-GPU implementation can not only lower the inter-process communication by reducing the number of processes but also explicitly utilize the NVLinks for the communication among the local GPUs of the process. We first describe our implementation (Section VI-A). We then profile the costs of the required communication on the node and between the nodes (Sections VI-B and VI-C). Finally, we show the performance of this multi-GPU implementation (Section VI-D).

A. Multiple GPUs Implementation

Here, we describe a new implementation of the solver that lets each process use multiple GPUs on the node to perform HiMV. To this end, the batches of the first type are executed

from\to	0	1	2	3
0	6.3 / 212.5 / NA	6.3 / 9.3 / 18.5	6.0 / 9.9 / 31.0	6.0 / 9.9 / 18.5
1	6.3 / 9.3 / 18.5	6.3 / 212.5 / NA	6.0 / 9.9 / 18.5	6.0 / 9.9 / 31.2
2	6.0 / 9.9 / 33.7	6.1 / 9.9 / 18.4	6.1 / 211.9 / NA	6.1 / 9.3 / 18.4
3	6.1 / 9.9 / 18.4	6.1 / 9.9 / 31.2	6.1 / 9.3 / 18.5	6.1 / 212.5 / NA

Fig. 12. Bandwidth (GB/s) for the GPU point-to-point data copy using `set` / `copy` / `peer` on a single node of Tsubame 3.

GPU	0	1	2	3
0	12.1 / 12.2 / 12.2	6.1 / 5.2 / 7.2	5.9 / 3.5 / 7.0	5.7 / 2.7 / 6.9
1	5.6 / 2.7 / 6.9	12.1 / 12.2 / 12.2	11.3 / 5.4 / 7.2	5.7 / 3.5 / 7.0
2	5.7 / 3.4 / 6.9	5.7 / 2.5 / 6.8	11.3 / 11.4 / 11.4	5.7 / 5.1 / 6.9
3	11.3 / 5.3 / 6.9	6.1 / 3.4 / 6.9	5.6 / 2.5 / 6.9	11.3 / 11.5 / 11.5

(a) `set` / `copy` / `peer` where (i,j)-th cell broadcasts to $i,\dots,j$-th GPUs.

GPU	(0)	(0,1)	(0,2)	(3,0)
GB/s	6.2 / 6.6	5.6 / 6.9	5.6 / 8.4	5.6 / 6.9

(b) `set` / `peer` with multiple GPUs/process: '(0)' uses one process per GPU, while the rest have one process per two GPUs with '(i,j)' using i-th and j-th GPUs for the first process.

Fig. 13. Bandwith (GB/s) for broadcasting from CPU to GPUs on a single node of Tsubame-3.

on the multiple GPUs in a round-robin fashion. Then, to avoid copying the vectors $\mathbf{t}^{(k)}$ among the local GPUs, the batch of the second type is executed on the GPU where the corresponding batch of the first type was executed. The vector operations are redundantly computed on all the GPUs to avoid the data copy between the local GPUs.

This multi-GPU implementation introduces two local communication phases where each process exchanges the data among the local GPUs:

1) **Local all-reduce**: The partial results of `HiMV` from all the local GPUs are gathered and summed on one of the GPUs. The resulting data is then copied to the CPU before the MPI call.

2) **Local broadcast**: After the MPI call, the CPU broadcasts the resulting global vector to all the local GPUs.

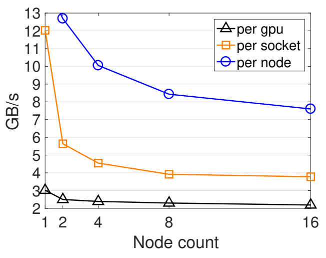
Though the matrix is distributed to maintain the load balances, there often exists load imbalance among the processes, and the imbalance may increase with the GPUs (e.g., due to the difference in the kernel performance with different sizes of the blocks distributed to each process). In addition, the data transfer to the GPUs may introduce different amount of idling time on the GPUs due to the hardware bottlenecks. Such load imbalances from the different phases of the iteration may be reduced by removing all the synchronization points between them. To this end, our multi-GPU implementation extensively uses the GPU streams and events so that the only synchronization points are those before and after the MPI communication. For instance, using one process per node on eight nodes of Tsubame-3, having the synchronizations can increase the iteration time by $15 \sim 20\%$ for the matrix `lms`, and the effects of the synchronizations may increase on a larger number of GPUs or with a larger load imbalance.

B. Inter-GPU Communication on a Node

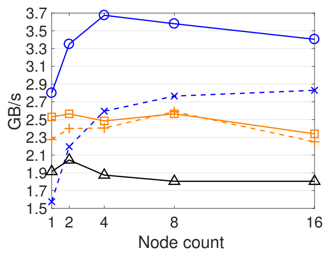
We now benchmark the cost of the local communication needed for our multi-GPU implementation: local all-reduce and local broadcast. We focus on Tsubame-3 since Reedbush-H has a simpler node architecture (e.g., each node has only two GPUs, and the CPU can broadcast to the GPUs in parallel using a different PCIe to each GPU).

In Figure 12, we look at transferring 8MB of data between a pair of GPUs for performing local all-reduce where we tested three implementations: the first one, called `set`, copies the data to the CPU's pinned memory using `cublasGetVector` before copying it to the target GPU using `cublasSetVectorAsync`, while the other two, referred to as `copy` and `peer`, directly copy the data between the GPUs using `cudaMemcpyAsync` and `cudaMemcpyPeerAsync`, respectively. The bandwidth is computed as the ratio of the data size sent between the GPUs over the total time needed for the communication. The figure shows that `set` obtained about a half of the observed peak bandwidth between the CPU and GPU since the data needs to go to the CPU before being copied to the target GPU. A slightly higher bandwidth was obtained using `copy`, especially for the pair of the GPUs that are connected through separate PCIe's to the CPU. This might be because we can pipeline the two phases of the data copy using two separate PCIe's: copying first from the source GPU to the CPU and then from the CPU to the target GPU. We did not pipeline the data copy for `set`, and hence, these two communication phases occur in sequence. We obtained much higher bandwidth (about half of observed GPU bandwidth) using `copy` on the same GPU because only the local data read and write on the GPU were needed. Finally, `peer` obtained higher bandwidth utilizing the NVLink between the GPUs. Moreover, compared with the bandwidth between GPU-0 and GPU-1, the bandwidth between GPU-0 and GPU-2 is doubled since these two GPUs are connected by two NVLinks (see Figure 4(b)).

Next, Table 13(a) shows the bandwidth for broadcasting the data from the CPU to the local GPUs where we again tested three configurations: the first configuration `set` broadcasts the data from the CPU to all the GPUs using `cublasSetVectorAsync` and one GPU stream per GPU, while the other two configurations `copy` and `peer` use `cublasSetVectorAsync` to copy the data only to the first GPU and then use `cudaMemcpyAsync` or `cudaMemcpyPeerAsync` to copy the data from the first GPU to the rest of the GPUs on different streams. The bandwidth is computed as the ratio of the data size over the time needed to complete the broadcast. With `set`, the amount of time needed to copy the data to both GPU-0 and GPU-2 was about the same as the time needed to copy the data only to GPU-0. On the other hand, the bandwidth halved when copying the data to GPU-0 and GPU-1. This is because the CPU is connected to GPU-0 and GPU-2 through two separate PCIe's, while the data transfer to GPU-0 and GPU-1 must go through the same PCIe (see Figure 4(b)). Hence, the data can be concurrently copied to GPU-1 and GPU-2 in parallel, while the data can be only transferred to GPU-0 and GPU-1, sequentially. The table also shows that using `copy` lead to slower data transfer since the data needs to go through the same PCIe from the source GPU (without NVLinks). On the other hand, using `peer`, we obtain more consistent bandwidth

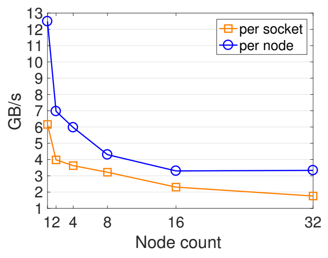


(a) Allgatherv on CPUs.

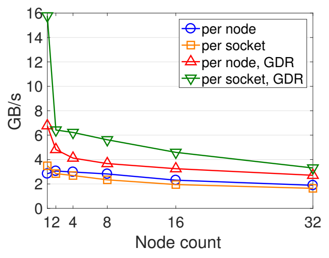


(b) Allgatherv on GPUs.

Fig. 14. Performance of `MPI_Allgatherv` on Tsubame-3.



(a) Allgatherv on CPUs.



(b) Allgatherv on GPUs.

Fig. 15. Performance of `MPI_Allgatherv` on Reedbush-H.

with different numbers of target GPUs since the data can be communicated between the GPUs using NVLinks in parallel. As expected, though the communication to some GPUs may complete faster than the others, each configuration achieved about the same overall bandwidth when it sent the data to all the GPUs regardless of the order the data is sent to the GPUs. However, the bandwidth was higher using `peer` than using `set`, which was faster than using `copy`.

In Figure 13(b), we again broadcast the data from the CPU to all the GPUs on the node. For these experiments, however, we used multiple MPI processes on the node and assigned different numbers of the GPUs to the process. We have studied two configurations: 1) with `set`, each process broadcasts the data from the CPU to all the GPUs and 2) with `peer`, the process only copies the data from the CPU to one GPU, and it then copies the data from the GPU to the other GPUs through NVLink. When each process has multiple GPUs with `peer`, it was important that each process communicates with a specific GPU so that we can reduce the contention on the PCIe connecting the CPU to the GPUs. For example, with both (0,1) and (0,2) configurations, Process-0 talks with GPU-0 while Process-1 talks with GPU-3. If Process-1 talks with GPU-1, there will be more stress on the PCIe connecting the CPU with these two GPUs. As we expected, `set` obtained about the half of the observed peak bandwidth in all the setups. We obtained higher bandwidth using `peer`, and the highest bandwidth of about 8.5 GB/s was obtained using the (0,2) configuration, utilizing the two NVLinks between the local GPUs (compared with 6.9 GB/s obtained to broadcast to all the GPUs with one process per node in Table 13(a)).

C. Inter-GPU Communication between Nodes

We now look at the cost of gathering a vector distributed over multiple nodes of Tsubame-3 and Reedbush-H. We designed the benchmark to reflect our `HiMV` implementation discussed in Section VI-A. In particular, for these experiments, we evenly distributed the vector $\mathbf{v}$ among the processes such that the j-th process has the local vector $\mathbf{v}_j$ of the length n_j. Then, each of the j-th process' local GPUs stores an n_j-length vector such that $\mathbf{v}_j$ is the sum of the vectors stored on the local GPUs (i.e., $\mathbf{v}_j = \sum_k \mathbf{v}_j^{(k)}$ where $\mathbf{v}_j^{(k)}$ is the vector on the k-th local GPU). Then for our benchmark, the process first sums the local vectors distributed among the GPUs on one of its local GPUs. Then, the resulting vector is copied from the GPU to the CPU's pinned memory before performing `MPI_Allgatherv`. After the MPI communication, each process broadcasts the global vector to its local GPUs. The motivation for having each process manage the multiple GPUs is to lower the inter-process communication cost by reducing the number of processes. However, there is the additional communication and computation needed to sum the local vectors distributed among the local GPUs.

For our benchmark on Reedbush-H, we tested two configurations 1) `per-node`: one process per node and two GPUs per process and 2) `per-socket`: one process per socket and one GPU per process, while on Tsubame-3, we had three configurations 1) `per-node` with four GPUs per process, 2) `per-socket` with two GPUs per process, and 3) `per-gpu` with one GPU per process and two processes per socket (`per-gpu` is equivalent to `per-socket` on Reedbush-H). For these experiments, we fixed the vector length to be 10^6 (i.e., 8MB of data) and increased the node count. We show the average time of two separate runs where each run computes the average time needed for 100 all-gather communication. The bandwidth is computed as the size of the global vector over the total communication time.

To benchmark the MPI's communication without GPUs, Figures 14(a) and 15(a) first show the performance of `MPI_Allgatherv` alone. We clearly see that MPI communication time may be reduced having a fewer process per node. Figures 14(b) then shows the performance with the GPUs on Tsubame-3, where we tested using either `copy` (dotted line) or `peer` (solid line) between the first GPU and the rest of the GPUs after the data is copied to the first GPU. Compared with the MPI communication alone, the inter-GPU communication took significantly longer due to the expensive data transfer between the CPU and GPUs through the PCIe. On a small number of nodes, the inter-GPU communication could be slower when each process manages multiple GPUs because of the overhead needed to sum the local vectors among the local GPUs. However, as the node count increases, the local vector shortens reducing this overhead, and at the same time, the inter-node communication starts to become more significant. As a result, the overall communication was often faster when there was a fewer processes on each node. Though the inter-GPU communication was faster using `per-socekt` on one

	`100ts`				`1ms`				`hum4`				`hum6`			
	n_d	iter	time (ms)	time/it	n_d	iter	time (ms)	time/it	n_d	iter	time (ms)	time/it	n_d	iter	time (ms)	time/it
per-gpu	2	15	56 (11)	3.7	4	14	410 (100)	29.3	2	73	1044 (141)	14.3	4	95	2396 (614)	25.2
	4	14	43 (11)	3.1	8	15	456 (183)	30.5	4	70	846 (192)	12.1	8	101	2260 (786)	22.4
	8	20	64 (26)	3.2	16	14	516 (251)	36.9	8	79	919 (333)	11.6	16	104	2705 (1233)	26.0
	16	19	81 (40)	4.3	32	14	712 (411)	50.9	16	78	992 (479)	12.7	32	101	3289 (1920)	32.6
per-socket	2	14	53 (12)	3.8	4	14	366 (72)	26.2	2	74	1010 (94)	13.7	4	76	1671 (187)	22.0
	4	13	43 (13)	3.3	8	14	306 (80)	21.9	4	72	693 (101)	9.6	8	94	1799 (452)	19.1
	8	15	46 (16)	3.1	16	15	341 (130)	22.8	8	62	649 (122)	10.5	16	99	1821 (518)	18.4
	16	19	74 (32)	4.6	32	15	433 (215)	28.9	16	77	860 (293)	10.8	32	104	2337 (811)	22.5
per-node	2	14	53 (9)	3.8	4	14	334 (42)	23.9	2	72	950 (38)	13.2	4	93	1932 (129)	20.8
	4	15	46 (10)	3.1	8	14	277 (49)	19.8	4	73	664 (57)	9.1	8	95	1531 (152)	16.1
	8	13	38 (10)	3.0	16	14	257 (60)	18.4	8	75	601 (65)	8.0	16	96	1445 (265)	15.1
	16	13	42 (12)	3.3	32	14	260 (95)	18.6	16	71	586 (99)	8.2	32	99	1594 (310)	16.1

Fig. 16. BiCG performance with multiple GPUs per process on Tsubame-3: the numbers in parentheses are the MPI time and n_d is the number of nodes.

node, `per-node` was faster on multiple nodes, reducing the inter-node communication cost.

Figure 15(b) shows similar results on Reedbush-H, along with the performance of the GPU-aware OpenMPI-GDR[1]. Without GDR, we need to copy the data to the CPU before calling MPI. With GDR, the data still goes out of the node through PCIe, but we can avoid the back-and-forth data copy between the CPU and switch (reducing the number of the data transfers through the PCIe by a factor of two). The GDR may also avoid some data copies between MPI's internal buffers. As a result, we observed that using `per-node` configuration, GDR obtained the speedups of about $2.4\times$ and $1.4\times$ on one and sixteen nodes, respectively. With `per-socket`, GDR obtained even greater speedups of about $4.5\times$ and $2.4\times$ on one and sixteen nodes, respectively. Also, due to the `per-node`'s overhead to sum the vectors among the local GPUs, GDR obtained a greater bandwidth using `per-socket` compared with `per-node`. However, as we increase the node count, the difference in the bandwidth was reduced (since the local overhead with `per-node` diminishes while having a fewer processes reduces the inter-process communication cost). Overall, with enough GPUs, the good combination seems to be using GDR among the processes and NVLink among the multiple GPUs on each process. Unfortunately, at the time of preparing this paper, the GPU-aware MPI through OmniPath was not supported on Tsubame-3.

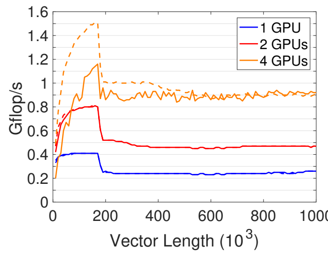
D. *Performance of Multi-GPU Implementation*

Figure 16 shows the effects of different process/GPU configurations on the BiCG performance. The solution time can be significantly reduced by assigning multiple GPUs to each process primarily because of the reduction in the MPI time. Overall, `per-node` was up to $2.7\times$ faster than `per-gpu`.

E. *Hierarchical Parallelization*

To reduce the sequential bottleneck, we developed another implementation that lets each process parallelize the vector operations among the local GPUs. This parallelization needs to be carefully designed since it introduces the communication among the local GPUs. For this, we parallelized the update of

[1]The `mpirun` options `--mca pml ^yalla --mca mtl ^mxm --mca coll ^hcoll --mca btl_openib_want_cuda_gdr 1 --mca mpi_warn_on_fork 0` is used to enable GDR.



(a) Strong-scaling of `dgemv` with a matrix of two vectors on one P100 GPU: the solid line sums the local vectors on the CPUs, while the dashed line accumulates them on the GPU using `MemcpyPeer`.

	scheme	
matrix	redundant	hierarchical
`hum4`	1.99 / 81	1.07 / 85
`hum6`	5.23 / 102	2.60 / 106
`100ts`	0.22 / 19	0.17 / 19
`1ms`	1.04 / 15	0.50 / 14
`8ms`	10.56 / 20	6.09 / 20
`20ms`	27.61 / 20	19.79 / 20

(b) Time in seconds / # iterations on 64 nodes with one process per node.

Fig. 17. Performance with hierarchical parallelization on Tsubame-3.

the solution vector $\mathbf{x}$ and the dot-products in Figure 2(a). In addition, for the experiments in the remaining of the paper, instead of using `cublasDdot` to compute the dot-products, we used the matrix-vector multiply `cublasDgemv`. For the tall-skinny matrices, `dgemv` may not be optimized as well as `ddot` is, but it allows us to merge multiple dot-products into a single kernel launch (e.g., $[\mathbf{r},\mathbf{v}]^T\mathbf{r}_0$, $[\mathbf{v},\mathbf{t}]^T\mathbf{t}$, or $[\mathbf{r}_0,\mathbf{r}]^T\mathbf{r}$ in Figure 2(a)). After the local `dgemv`, the pair of GPUs independently accumulate their results using NVLink (e.g., on Tsubame-3, GPU-0 and GPU-2, and GPU-1 and GPU3) before sending their accumulated result to the CPU and computing the final result. Figure 17(a) shows the performance of `dgemv` using four P100 GPUs on one node of Tsubame-3, and demonstrates that the performance can be improved using the multiple GPUs. Figure 17(b) then shows that this "hierarchical" parallelization can reduce the sequential bottleneck and improve the solver performance. We also investigated keeping the scalars on the GPUs. This however leads to a few scalar operations using BLAS on the GPU (i.e., the computation of α, ζ, β, and γ), and often resulted in a lower solver performance.

VII. Hiding Inter-GPU Communication

Two approaches have been developed to address the inter-process communication cost of the Krylov solvers. The first approach is the "communication avoiding" (CA) that is based on the s-step variant of the Krylov solvers [10], [11] and redesigns the algorithm to communicate less by generating

#bytes	$t_{\text{ovrl}}[\mu\text{sec}]$	$t_{\text{pure}}[\mu\text{sec}]$	$t_{\text{CPU}}[\mu\text{sec}]$	overlap[%]
16384	2386.29	1945.92	2374.40	81.45
32768	2908.06	2786.06	2849.83	95.72
65536	4883.94	3933.52	4867.52	80.47
131072	11213.05	10552.64	11171.78	94.09
262144	19919.10	19834.93	19845.31	99.58
524288	38677.25	31071.01	38540.00	80.26

Fig. 18. Results of Intel MPI Benchmark for `MPI_Iallgatherv` on 16 nodes of Tsubame-3 with 4 processes per node.

	`100ts`			`338ts`			`hum4`		
n_g	block	pipe1	pipe2	block	pipe1	pipe2	block	pipe1	pipe2
1	18.4	19.2	19.3				−−	−−	−−
4	10.2	10.0	9.4	34.6	33.8	31.6	34.3	32.1	30.4
8	9.1	8.9	7.6	31.5	30.1	25.6	30.5	28.1	24.3
16	8.8	8.6	6.7	30.2	28.6	22.0	30.1	28.0	21.1
32	10.3	10.1	6.8	31.7	30.8	21.8	30.9	28.3	19.7

Fig. 19. Time per iteration (ms) with different GPU count, n_g, on Tsubame-3.

a set of s basis vectors at a time. Though it has the potential to reduce the communication latency cost by the factor of s, HACApK requires the global vector for each application of `HiMV`. Hence, in order to avoid the communication, each process would require to redundantly compute the global `HiMV`, falling back to the sequential performance [12].

The second approach is "pipelining" that redesigns the algorithms to hide the cost of communication by exploiting non-blocking communication and pipelining the iterations [13], [14]. Though simply hiding the communication only leads to the maximum speedup of $2\times$, pipelining the communication may leads to greater speedups. The technique is designed to hide the all-reduce communication required for the dot-products behind the local computation of the sparse matrix-vector multiply. In contrast, HACApK avoids the all-reduces and its only inter-process communication is the all-gather after `HiMV`. Unfortunately, the all-gather cannot be overlapped with the local computation of `HiMV` since they are on the critical path of the algorithm. Nevertheless, we adapt the pipelining technique to hide the all-gather behind the vector operations.

Figure 2(b) shows the variant of BiCG. Though this variant introduces additional computation (i.e., increases the number of vector operations from 12 to 20), it hides the all-gather communication behind the local dot-products. We looked at two variants: 1) `pipe1` hides the vector copy from the GPU to the CPU and 2) `pipe2` hides `MPI_Allgatherv` between the CPUs. For these experiments, we used MPICH version 3.2 that implements `MPI_Iallgatherv` using TCP and IP-over-Infiniband. We configured MPICH with `--enable-threads=multiple` and initialized the MPI library using `MPI_THREAD_MULTIPLE`. We chose to use MPICH mainly because of its `MPI_Iallgatherv`'s capability to overlap the communication with the computation (see Figure 18). To show the potential of hiding the communication, Figure 19 shows its effects on the performance of our single-GPU implementation of the solver. The greater performance improvements were obtained using `pipe2` with the maximum speedup of about $1.57\times$.

VIII. Conclusion

In this paper, we carefully ported the hierarchical-matrix BiCG solver onto GPU clusters. We investigated several techniques to improve the performance of the batched GPU kernel (sorting the tasks, dividing them into multiple batches of different batch sizes, and using GPU streams to execute multiple batches in parallel). We hope that these techniques were integrated into the future generation of the batched kernel along with a runtime analysis and tuning. We have also studied several techniques to reduce the inter-GPU communication. As the heterogenous node architecture becomes increasingly complex and the cost of the inter-GPU communication increases, these techniques likely become critical to many applications running on GPUs. We have observed a great potential of the GPU-aware MPI that complements our studies, and we plan to investigate its performance on different architectures (e.g., non-blocking collective on Summit). We would also like to investigate other algorithmic techniques to address the communication costs (e.g., 2D partitioning, empty off-diagonal blocks with rank zero). Though we ported only the linear solver onto the GPUs, we plan to port other parts of the simulation (e.g., generation or compression of the matrix).

References

[1] T. Iwashita, A. Ida, T. Mifune, Y. Takahashi, Software framework for parallel BEM analyses with H-matrices using MPI and OpenMP, in: Proceedings of the International Conference on Computational Science, 20017, pp. 12–14.

[2] A. Ida, T. Iwashita, T. Mifune, Y. Takahashi, Parallel hierarchical matrices with adaptive cross approximation on symmetric multiprocessing clusters, Journal of Information Processing 22 (2014) 642–650.

[3] H. A. van der Vorst, Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems, SIAM. J. Sci. and Stat. Comput. 13 (2) (1992) 631–644.

[4] A. Ida, T. Iwashita, T. Mifune, Y. Takahashi, Variable preconditioning of Krylov subspace methods for hierarchical matrices with adaptive cross approximation, IEEE Transactions on Magnetics 52 (3) (2015) 1–1.

[5] `http://www.t3.gsic.titech.ac.jp/node/6`.

[6] `http://www.cc.u-tokyo.ac.jp/system/reedbush/reedbush_intro.html`.

[7] T. Dong, A. Haidar, S. Tomov, J. Dongarra, Optimizing the SVD bidiagonalization process for a batch of small matrices, Procedia Computer Science 108 (Supplement C) (2017) 1008 – 1018, international Conference on Computational Science, ICCS 2017.

[8] A. Abdelfattah, A. Haidar, S. Tomov, J. Dongarra, Novel hpc techniques to batch execution of many variable size blas computations on gpus, in: Proceedings of the International Conference on Supercomputing, 2017, pp. 5:1–5:10.

[9] A. Abdelfattah, A. Haidar, S. Tomov, J. Dongarra, Performance, design, and autotuning of batched GEMM for GPUs, in: Proceedings of the International Supercomputing Conference (ISC) High Performance, 2016, pp. 21–38.

[10] M. Hoemmen, Communication-avoiding Krylov subspace methods, Ph.D. thesis, EECS Dep't, Univ. of Calif., Berkeley (2010).

[11] E. Carson, N. Knight, J. Demmel, Avoiding communication in two-sided Krylov subspace methods, Tech. Rep. UCB/EECS-2011-93, EECS Dept., U.C. Berkeley (2011).

[12] M. Mohiyuddin, M. Hoemmen, J. Demmel, K. Yelick, Minimizing communication in sparse matrix solvers, in: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, 2009, pp. 36:1–36:12.

[13] P. Ghysels, T. Ashby, K. Meerbergen, W. Vanroose, Hiding global communication latency in the GMRES algorithm on massively parallel machines, SIAM J. Sci. Comput. 35 (2013) C48–C71.

[14] S. Cools, W. Vanroose, The communication-hiding pipelined BiCGstab method for the parallel solution of large unsymmetric linear systems, Parallel Comput. 65 (C) (2017) 1–20.