Vol.2009-HPC-121 No.2
2009/8/4

A Memory-Efficient Algorithm and Its Implementation of Variable-Size All-to-All Communication

Bingbing Zhuang,[†2] Hiroshi Nakashima[†1]
and Hiroshi Nagamochi[†2]

This paper proposes a memory-efficient algorithm of variable-size all-to-all communication based on *bitonic sort* and *in-place merge*. The algorithm takes $O(N \log^2 P)$ computation and communication time and requires merely $O(P)$ extra space for the communication among P processes having N elements of data to be exchanged for each. We also discuss another algorithm which takes $O(MP \log P)$ time where M is the size of the largest chunk which a process must send to another process and thus is expected to be $O(N/P)$ in most practical applications to make the time complexity $O(N \log P)$. We implemented the first sort-based algorithm to show it works with reasonable efficiency.

1. Introduction

One of important motivations of parallel high-performance computing is, besides obvious expectation of parallel speedup, to enlarge the size of problems to be solved. This means that parallel algorithms must be aware of memory efficiency as well as timing efficiency. In a sense, memory efficiency is more critical than timing efficiency because we may allow 10 % execution time extension while 10 % memory size excess cannot be acceptable or we have to pay a huge memory/disk swap cost.

Memory efficiency of collective communications sometimes governs that of application programs using them. For example, if a program has a large distributed array which dominates the memory requirement of the program and we need to perform FFT on the array, all-to-all communication for transposition must be performed with memory awareness. Bad news to MPI users is that `MPI_Alltoall()` does not have *in-place* option[⋆1] to replace the data in send-buffer with those received. Therefore we need implement a hand-made version of all-to-all to restrict additional space to $O(1)$ or waste memory space as large as the array.

The variable-size variance of all-to-all communication, e.g., `MPI_Alltoallv()`, is also useful for parallel applications especially for load distribution/exchange among parallel processes. For instance, our particle-in-cell simulator[4)] needs an all-to-all distribution of particle subsets to initialize the simulation and/or to correct unacceptable load imbalance. Unfortunately, devising and implementing a memory efficient variable-size all-to-all is not trivial work. That is, in a fixed-size all-to-all, a process p can exchange a fixed-size chunks with another process q to place the chunks to their destinations, or can shift chunks along a rings connecting processes. However, these simple techniques are not applicable for the variable-size version because the pair of the chunks or those on the circular shift path may have arbitrary variable sizes.

The work presented paper aims at devising memory efficient and reasonably time efficient all-to-all algorithms applicable to memory intensive parallel applications including our particle-in-cell simulator by which we are motivated to pursue the work. In the rest of the paper, after defining the problem to be solved, we discuss two algorithms, one based on bitonic sort and the other with pairwise exchange. We also show our experiment results on the first algorithms to evidence its reasonable timing efficiency.

2. Problem Definition and Assumptions

For the sake of explanation simplicity[⋆2], we define our variable-size all-to-all communication problem as follows. Let $s(i, j)$ be a sequence of data elements $a_1^{ij}, \ldots, a_n^{ij}$ which a parallel process $i \in [1, P]$ accommodates at initial and a process $j \in [1, P]$ finally stows in its memory. The initial layout of $s(i, 1), \ldots, s(i, P)$ in the process i is the concatenation of them in this order denoted by $s(i, 1) \bullet \cdots \bullet s(i, P)$. The *stable* solution for the process j is to have the

†1 ACCMS, Kyoto University
†2 Graduate School of Informatics, Kyoto University

⋆1 Good news is that in-place option is expected to be incorporated a (near) future version of MPI specifications.
⋆2 Also for the easiness of our implementation, so far.

concatenation $s(1,j) \bullet \cdots \bullet s(P,j) \equiv S(j)$, while *unstable* one is a permutation of data elements in $S(j)$.

Let $|s(i,j)|$ be the length (or number of data elements) of the sequence $s(i,j)$ and N be that of the initial and final concatenation of the sequences regardless the process accommodating it. That is, for all $i \in [1,P]$, the following is satisfied.

$$\sum_{k=1}^{P} |s(i,k)| = \sum_{k=1}^{P} |s(k,i)| = N.$$

The memory overhead of an algorithm is the maximum space required in addition to that for N elements throughout the procedure to solve the problem. It is obviously desirable that the overhead is $O(1)$ but $O(P)$ overhead is acceptable. In fact, since a data element may be anything and thus may tell us neither its initial housing process, its initial location in the process, nor its final destination process, it is almost inevitable that an array containing $|s(i,1)|, \ldots, |s(i,P)|$ is given as the input for the process i besides the sequences $s(i,1) \bullet \cdots \bullet s(i,P)$, as `MPI_Alltoallv()` requires an input argument `sendcounts[`P`]`[⋆1].

The time complexity of an algorithm is determined by two factors, computation cost and communication cost. Computation cost is mainly for memory operations to manipulate sequences and elements in them and, of course, is calculated taking parallel execution into account. As for communication cost, we simply assume that a transmission of n data elements from a process p to a process q takes $O(n)$ time. That is, we neglect the constant overhead for the communication and the latency possibly depending on the *distance* between them in the process network. We also assume that processes are connected with a non-blocking network through which communications between any number of non-overlapping pairs can be performed simultaneously without any performance interference. Finally, if a time complexity has factors of N and P with the same degree, we neglect P to reduce, for example, $N+P$ to N because $N \gg P$.

⋆1 The argument `recvcounts` is not necessary to be given explicitly. Since we assume the fixed order of initial and final sequences as the fixed-size `MPI_Alltoall()` does, `sdispls` and `rdispls` are also unnecessary. If it is required to specify the permutation of the sequences by the displacement arguments, pre- and post-processing can be implemented using a in-place sort with the in-place merger discussed later.

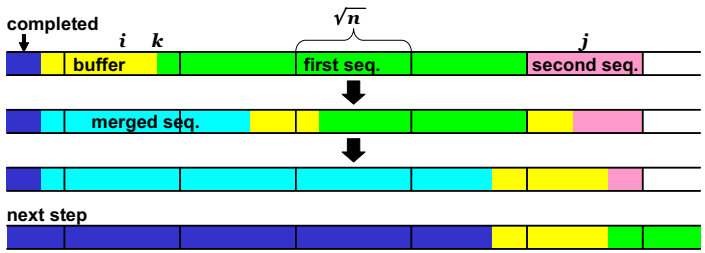

Fig. 1 Progress of merge process.

3. Algorithms

3.1 Fundamental Means

For designing of memory efficient algorithms, we have two fundamental means which requires merely $O(1)$ extra space regardless the size of data to operate on. The first one is inter-process swap namely $swap(i,j,s_i,s_j)$ to exchange sequences s_i and s_j of size $n = |s_i| = |s_j|$ between the process pair i and j. This operation can be straightforwardly performed by `MPI_Sendrecv_replace()` which is expected to use a fixed size data buffer to temporarily store sending data before directly receiving data into its `sendbuf` (or to store receiving data which is then copied to `sendbuf`). Since we have to repeat send/receive communications if the buffer is smaller than the size of n elements, a large buffer of a few mega-bytes, for example, is desirable but the size is still $O(1)$ and is much smaller than N.

The second mean $merge(s_1,s_2)$, which operates on the concatenation of ordered sequences $s = s_1 \bullet s_2$ to merge the sequences replacing them, is more powerful and complicated than the *swap*. The algorithm, invented by Huang and Langston[2], takes $O(|s|)$ time while requiring $O(1)$ extra space for stable merge. The outline of the algorithm is as follows.

(1) Let $n = |s|$ and split s into $\sqrt{n}$ blocks $b_1, \ldots, b_{\sqrt{n}}$ so that each block has $\sqrt{n}$ elements.

(2) Rotate a part of s so that $b_{\sqrt{n}}$ has $\sqrt{n}$ largest elements, and then Rotate s so that $b_{\sqrt{n}}$ is placed leftmost and thus becomes b_1. Then sort blocks $b_2, \ldots, b_{\sqrt{n}}$ using the last elements of block as keys by a stable sorting algorithm with $O(1)$ extra space, for example, by selection sort taking

$O((\sqrt{n})^2) = O(n)$ time.

(3) Let $i = 2$ and $k = 1$, and repeat following steps (4) to (6) until one of termination conditions is satisfied.

(4) Find a block b_j block such that $i \leq j$ and the block is the leftmost one whose last element is larger than the first element of b_{j+1}. If such a block is not found, terminate the loop and go to the step (7).

(5) Merge two sequences, one is from k-th element of b_i block to the last element of b_j and the other is b_{j+1}, using $\sqrt{n}$ elements preceding the left sequence to be merged as a buffer. That is, repeat the exchange of the leftmost element of the buffer and the larger one of two leftmost elements of the sequences to be merged, until the last element of b_j is exchanged as shown in **Fig. 1**. The elements in the buffer were originally in $b_{\sqrt{n}}$ and are moved so that they follow the merged sequence.

(6) Let $i = j + 1$ and k be the index of the leftmost element staying in b_{j+1}.

(7) Rotate the subsequence from the head of the buffer to the tail of the sequence so that the buffer is placed rightmost and becomes $b_{\sqrt{n}}$ again. Then sort the buffer using (e.g.) selection sort taking $O((\sqrt{n})^2) = O(n)$ time again.

Note that each of steps (2) and (7) takes $O(n)$ as well as the loop from (4) to (6) as the whole. Also note that the implementation difficulty of this algorithm lies in the technique to use the largest $\sqrt{n}$ elements as the buffer keeping stability, and this difficulty is drastically reduced if we allow $O(\sqrt{n})$ extra space which is practically acceptable because the space is as large as about 30 thousands elements for the sequence having 1 billion elements[⋆1].

3.2 Algorithm Based on Bitonic-Sort

Here we describe the first algorithm based on bitonic sort. An all-to-all communication can be considered as a parallel sorting with the following total ordering of data elements.

$$a_{k_1}^{i_1 j_1} < a_{k_2}^{i_2 j_2} \iff j_1 < j_2 \ \vee \ (j_1 = j_2 \ \wedge \ i_1 < i_2) \ \vee \ (j_1 = j_2 \ \wedge \ i_1 = i_2 \ \wedge \ k_1 < k_2).$$

That is, the sorting with ordering above results in the sequence $S(1) \bullet \cdots \bullet S(P)$. Since we assume $\sum_{i=1}^{P} |s(i,j)| = N$ for all j, each part of the sorted sequence $S(j)$ should reside in the process j if we arrange that j has the j-th subsequence N long.

Bitonic sort[1)] is one of efficient and easily-parallelizable sorting algorithms. It operates on a bitonic sequence being the concatenation of an ascending sequence and a descending one, or a circular rotation of this fundamental sequence. The lower and upper halves of a bitonic sequence are easily found by scanning its first and second halves. More specifically, for a bitonic sequence $a_1, \ldots, a_{2n}$, the *crossing index* i such that $a_i < a_{2i} \leftrightarrow a_{i+1} \geq a_{2i+1}$ tells us the lower (or upper) half is the concatenation of $a_1, \ldots, a_i$ and $a_{n+i+1}, \ldots, a_{2n}$ while upper (or lower) half is $a_{i+1}, \ldots, a_{2n+i}$, if $a_i < a_{2i}$ (or $a_i \geq a_{2i}$). Since it is assured that the lower and upper halves are bitonic sequences again, iterative splitting of a bitonic sequence $a_1, \ldots, a_{2^k n}$ gives us subsequences $s_1, \ldots, s_2^k$ such that $|s_i| = n$ for all i, and $a_k \leq a_j$ for all $1 \geq i < j \leq 2^k$, $a_k \in s_i$ and $a_j \in s_j$.

Parallel implementation of bitonic sort with $P = 2^p$ processes consists of p phases (calls of procedure *bitonic_sort*()) as shown in **Figure 2** and **3**. The k-th phase starts from 2^{p-k} bitonic sequences each of which lies across 2^k processes, then each sequences is split k times so that each process has a bitonic sequence whose ascending and descending subsequences are merged to have a ascending or descending sequence. Note that *local_sort*() to make ascending (if *UP*) or descending (if *DOWN*) is unnecessary for our application to all-to-all because the initial sequence $s(i,1) \bullet \cdots \bullet s(i,P)$ is an ascending sequence in our ordering definition[⋆2].

The procedure *bitonic_split*() is for splitting a bitonic sequence lying across processes $i, \ldots, i + 2^k - 1$ performed by $swap(j, j + 2^{k-1}, s'_j, s'_{j+2^{k-1}})$ for all j such that $i \leq j \leq i + 2^k - 1$, where s'_j is a leading or trailing subsequence of the sequence s_j which j accommodates and is defined follows. Let j' such that $i \leq j' \leq i * 2^k - 1$ and the crossing index lies in the process j', A be true iff the crossing index on a ascending sequence, and L be true iff j needs lower half of

⋆1 Therefore our implementation uses $O(\sqrt{n})$ buffer, so far.

⋆2 In our implementation, we omit reversing $s(i,1) \bullet \cdots \bullet s(i,P)$ for even number processes but give the sequence an annotation that it should be considered as reversed.

```
procedure sort(i, p) begin
  if i mod 2 = 1 then local_sort(UP);
  else                local_sort(DOWN);
  for k = 1 to p do begin
    up ← (i − 1) mod 2^k < 2^(k−1);
    bitonic_sort(i, k, up);
  end
end
procedure bitonic_sort(i, k, up) begin
  for j = k − 1 downto 0 do begin
    lower ← up;
    if i ≤ 2^j then bitonic_split(i + 2^j, lower);
    else            bitonic_split(i − 2^j, ¬lower);
  end
  bitonic_merge(up);
end
```

Fig. 2 Pseudo code of parallel bitonic sort.

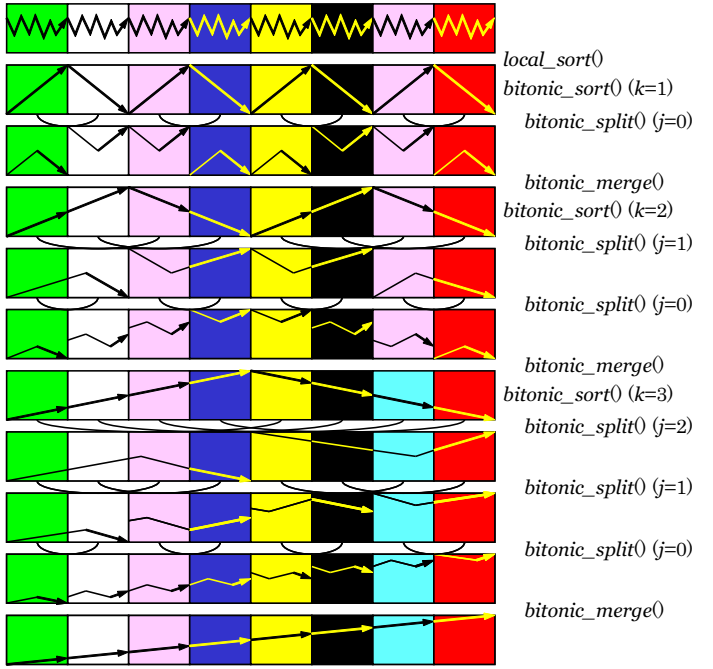

Fig. 3 Progress of bitonic sort with 8 processes.

the bitonic sequence. Then s'_j is defined as follows.

- s_j itself if $j \neq j'$ and $(j' < j \leftrightarrow A) \leftrightarrow L$.
- Leading (or trailing) subsequence of s_j preceding (or following) the crossing index on j (i.e., $j = j'$) if $\neg(A \leftrightarrow L)$ (or $A \leftrightarrow L$).
- Nothing if $j \neq j'$ and $(j' > j \leftrightarrow A) \leftrightarrow L$.

The procedure *bitonic_merge*() is for merging a bitonic sequence on a process performed by *merge*() after rotating the sequence if necessary.

Since the computation and communication cost of *swap*() is $O(N)$ and the computation cost of *merge*() is also $O(N)$, the total time complexity of the algorithm is $O(N \log^2 P)$. As for the space complexity, we just need an array of $2P + 1$ whose k-th element has the number of elements of k-th *chunk* in the sequence which is a subsequence of $S(i)$ for some i. That is, the sequence consists up to three subsequences each of which is represented by the destination of the first chunk, the number of chunks and the number of data elements in the subsequence. Therefore, the space complexity is $O(P)$.

3.3 Algorithm with Pairwise Exchange

The second algorithm is similar to pairwise exchange for fixed-size all-to-all, but the exchanging is *asymmetric*. That is, a pair of processes i and j such that $i < j$ performs $swap(i, j, s_i, s_j)$ so that i gets a part of $S(i)$ which j has but i gives j *arbitrary* data elements which is not necessary a part of $S(j)$. With this asymmetric exchange, a process $i \in [1, P]$ will have $S(i)$ or some permutation of its member elements by the following $P - 1$ communications whose global view for $P = 8$ is shown **Fig. 4**.

(1) Repeat the following for $j = 1$ to $i - 1$. Concatenate all subsequence of $S(j)$ residing in i to make s_i and do $swap(i, j, s_i, s_j)$ to get a sequence s_j.

(2) Repeat the following for $j = i + 1$ to P. Assemble a sequence s_i not having

$(1,2)$						
$(1,3)$						
$(1,4)$	$(2,3)$					
$(1,5)$	$(2,4)$					
$(1,6)$	$(2,5)$	$(3,4)$				
$(1,7)$	$(2,6)$	$(3,5)$				
$(1,8)$	$(2,7)$	$(3,6)$	$(4,5)$			
	$(2,8)$	$(3,7)$	$(4,6)$			
		$(3,8)$	$(4,7)$	$(5,6)$		
			$(4,8)$	$(5,7)$		
				$(5,8)$	$(6,7)$	
					$(6,8)$	
						$(7,8)$

Fig. 4 Global veiw of pairwise exchange communications.

any elements in $S(i)$ to be swapped with s_j which should be a sequence of elements in $S(i)$, and then do $swap(i, j, s_i, s_j)$.

In the algorithm, we have a free hand to compose s_i in the step (2). One reasonable way is to compose it by elements in $S(i+1)$, then those $S(i+2)$ if not suffice, and repeat this process until $|s_i| = |s_j|$.

The time complexity of this algorithm is hard to analyze. Bad news is the worst case complexity is $O(NP)$ in some extreme cases. For example, if the process 1 initially has $S(P)$ and other processes $i \in [2, P]$ have $S(i-1)$, the execution is serialized with $P-1$ communications of $swap(i, i+1, S(P), S(i))$. In practice, however, it is expected that initial setting is more moderate and the time complexity is $O(MP \log P)$ where M is the maximum size of swapped chunks in each *row* in Fig. 4 and is expected to be $O(N/P)$ to make the complexity $O(N \log P)$.

The $\log P$ factor of the time complexity is to merge received sequences. A process i should perform merging when it does $swap()$ with j such that $j < i$ if i has already received sequences in $S(j)$. The process i also needs to merge received sequences in $S(k)$ such that $k > i$ when it starts $swap()$ with the process $i+1$ if we adopt the reasonable way to compose s_i as discussed above. The number of merging operations and the size of sequences to be merged are expected to be small enough to make the time complexity $O(N \log P)$ but they could be too large resulting in $O(NP)$ complexity. In addition, the size of data structures to maintain the received sequences is expected to be $O(P)$ but can be $O(P^2)$ especially when we need the stability.

To summarize, this algorithm is faster than that based on bitonic sort in usual cases, but could be slower and inefficient with respect to the memory overhead depending on the initial setting. Therefore, we have to switch two algorithms by examining the initial setting. The implementation of this pairwise exchange algorithm is left for our future work together with the combination of two algorithms and switching criteria.

4. Experiments

We implemented the algorithm based on bitonic sort on our T2K Open Supercomputer[3] using Fujitsu's C compiler version 3.0 and MPI library version 3.0. We measured the performance with $N = 2^24$ data elements of 16 byte, or 256 MB memory space, for each process. The number of processes is set to 2^p varying p from 1 to 8 to have $P = 2$ to 256, and each process is allocated on a core of quad-core Opteron.

As for the initial setting of $s(i, j)$ for the process i, we examined the following four cases.

case-1: $|s(i,j)| = N/P$

case-2: $|s(i,j)| = \begin{cases} 0 & 0 \le (j-i) \bmod P < P/4 \\ N/P & P/4 \le (j-i) \bmod P < P-1 \\ N/4 & (j-i) \bmod P = P-1 \end{cases}$

case-3: $|s(i,j)| = \begin{cases} 0 & 0 \le (j-i) \bmod P < P/2 \\ N/P & P/2 \le (j-i) \bmod P < P-1 \\ N/2 & (j-i) \bmod P = P-1 \end{cases}$

case-4: $|s(i,j)| = \begin{cases} 0 & (j-i) \bmod P \ne P-1 \\ N & (j-i) \bmod P = P-1 \end{cases}$

Finally, we also measured the performance of `MPI_Alltoallv()` for all cases as reference.

Table 1 Execution time in second of bitonic-sort based algorithm and `MPI_Alltoallv`.

	bitonic-sort based				`MPI_Alltoallv`			
#proc	case-1	case-2	case-3	case-4	case-1	case-2	case-3	case-4
2	2.0	—	—	1.1	0.6	—	—	0.8
4	4.5	—	5.1	2.0	1.0	—	1.4	0.8
8	8.4	10.2	10.2	4.1	1.3	1.4	1.4	1.0
16	15.5	19.6	19.3	10.7	2.2	2.4	2.1	1.7
32	21.1	29.1	28.8	15.1	1.7	2.2	3.3	1.7
64	30.6	44.0	42.5	26.9	2.5	2.8	3.7	1.6
128	39.2	57.5	59.1	39.7	2.7	3.2	3.8	1.4
256	52.9	75.0	78.6	54.9	3.5	3.9	4.5	1.4

The measured execution times are shown in **Table 1**. Although execution times vary depending on the initial settings, they are fit well to the time complexity $O(N \log^2 P)$. In fact, if we can neglect the execution time of *merge*(), the time for one *swap*() is in the range 1.4–1.5 s in case-1, 1.7–2.1 s in case-2, 1.7–2.2 s in case-3, and 0.7–1.5 s in case-4. These numbers mean that the throughput of one *swap*() is 120–380 MB/s, which is not excellent but reasonable.

The absolute execution time up to 78.6 s for 256 process in case-3 is much larger than that of `MPI_Alltoallv()` which takes up to 4.5 s, but is acceptable for many purposes including our motivated example of load (re-)distribution in particle-in-cell simulation. Note that the execution time of one minute or so is significantly shorter than a straightforward solution relying on (local) disk storage, which might take 6–7 minutes for our 16-core nodes providing the effective throughput for swap-in/out of the node disk is 20 MB/s. Finally, the execution time of 4096 processes is expected to be about 2.5 minutes which is still significantly better than the solution with disk storage.

5. Conclusion

In this paper, we discussed two memory-efficient algorithms of variable-size all-to-all communication. The first algorithm based on bitonic sort takes $O(N \log^2 P)$ computation and communication time with $O(P)$ extra memory space. The second algorithm with asymmetric pairwise exchange will be faster than the first in usual cases because its $O(MP \log P)$ time complexity is expected to be $O(N \log P)$ if the data size M of each exchange is $O(N/P)$.

We implemented the first algorithm on our T2K Open Supercomputer and measured its performance using up to 256 CPU cores. This evaluation confirmed that the execution time is proportional to $N \log^2 P$ and the absolute value up to 78.6 s with 256 MB data for each of 256 processes is acceptably small.

Our urgent future work is to analyze the behavior of the second pairwise exchange algorithm in detail to find criteria to bound its time complexity to $O(N \log N)$ and memory overhead to $O(P)$. Then we will implement a combined algorithm to switch two algorithms according to the criteria.

Acknowledgments A part of this research work is pursued as a Grant-in-Aid Scientific Research #20300011 supported by the MEXT Japan.

References

1) Batcher, K.E.: Sorting Networks and their Applications, *Proc. AFIPS Spring Joint Computer Conf.*, Vol.32, pp.307–314 (1968).
2) Huang, B.-C. and Langston, M.A.: Fast Stable Merging and Sorting in Constant Extra Space, *Computer J.*, Vol.35, No.6, pp.643–650 (1992).
3) Nakashima, H.: T2K Open Supercomputer: Inter-University and Inter-Disciplinary Collaboration on the New Generation Supercomputer, *Intl. Conf. Informatics Education and Research for Knowledge-Circulating Society*, pp.137–142 (2008).
4) Nakashima, H., Miyake, Y., Usui, H. and Omura, Y.: OhHelp: A Scalable Domain-Decomposing Dynamic Load Balancing for Particle-in-Cell Simulations, *Proc. Intl. Conf. Supercomputing*, pp.90–99 (2009).