Vol.2010-HPC-124 No.9
2010/2/22

An Efficient Vector Transfer for Sparse Matrix-Vector Multiplication on Distributed Memory Systems

Hiroshi Nakashima,[†1] Toshiyuki Fukuhara[†2]
and Takeshi Iwashita[†1]

This paper proposes an efficient algorithm to exchange fragments of a vector distributed among processes which repeatedly perform sparse matrix-vector multiplication in parallel and in a block-decomposed manner. The idea to reduce the communication cost for the exchange is to *combine* non-contiguous fragments of a vector residing in a process and required by another process to multiply the fragments by the submatrix in the receiver process. That is, the sender may send fragments and *gaps* between them forming a larger fragment, rather than sending each fragment individually or packing fragments into a single vector before sending. The key feature of our algorithm is to find the *optimum assortment* of combining, individual sending and packing based on dynamic programming from given fragments and non-linear cost functions. Our preliminary evaluation with artificially generated sparse matrices shows the optimum assortment is up to 1.5 times as fast as simple packing.

1. Introduction

Sparse matrix-vector multiplication is a frequently used kernel operation in numerical and scientific programs including those implementing iterative methods to solve linear equations and to find eigenvalues. These iterative methods typically repeats $\boldsymbol{w}_{i+1} = \boldsymbol{A}\boldsymbol{v}_i$ with a fixed sparse matrix $\boldsymbol{A}$ and vectors $\boldsymbol{v}_i$ and $\boldsymbol{w}_{i+1}$ from which the multiplier vector of the next iteration step $\boldsymbol{v}_{i+1}$ is derived.

Parallelizing the programs and methods with sparse matrix-vector multiplication of large dimension is inevitable not only to accelerate them but also to simply accommodate the large size matrices and vectors in a computing system. The latter requirement usually leads us to decompose the matrix and vectors into submatrices and subvectors to make each of them reside in a portion of distributed memory and thus to assign the computation of a submatrix and subvectors to the process corresponding to the memory portion. For example, a row-wise block-decomposition assigns a set of contiguous rows of $\boldsymbol{A}$ to a process which should naturally have the corresponding part of the product vector $\boldsymbol{w}$. The process may also have a part of the multiplier vector $\boldsymbol{v}$ rather than the whole of it exploiting the sparseness of the matrix $\boldsymbol{A}$. That is, the process only requires a part of components of $\boldsymbol{v}$ which corresponds to the non-zero entries of $\boldsymbol{A}$.

The problem in giving a process its necessary components of $\boldsymbol{v}$ is that the components can be distributed in many processes because a component is usually derived from the corresponding component of $\boldsymbol{w}$ by the process having it. A simple but spatially and temporally inefficient solution is to use an all-gather type communication so that every process is given the whole of $\boldsymbol{v}$ regardless of its necessity. More sophisticated and efficient solution is to analyze the matrix $\boldsymbol{A}$ prior to the iterative multiplications to have a kind of flow graph representing every sender/receiver pair for each subset of $\boldsymbol{v}$'s components which the receiver requires and the sender has.

Since the $\boldsymbol{v}$'s components which a receiver needs are not necessary to be contiguous but may form a set of *fragments*, we have to determine how a sender sends the fragments to the receiver. A simple way is to send each fragment individually potentially causing a flood of small messages. Another simple tactics is to *pack* the components into a buffer for sending to minimize the number of messages paying the cost of copying. In addition, we found the third option in which fragments are *combined* together with the gaps between them to form a large message without copying.

Since the message transfer time with each of three tactics above depends on the number of fragments and the size of each fragment and each gap, the best solution is an *assortment* of the tactics applied to sets of fragments. For example, the individual sending will be apply to large fragments, while packing and combining should have advantage for a series of smaller fragments separated by large and small gaps respectively.

Our primary contribution described in this paper is the proposal of an algorithm based on dynamic programming to find the *optimum* assortment, which is

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

discussed in Section 3 after we define the problem more formally in Section 2. The implementation of the algorithm and its performance are then shown in Section 4. Finally we conclude the paper in Section 5.

2. Problem Definition

Let $\boldsymbol{A}$ be a sparse square matrix of $N \times N$ having entries $a(i,j)$ $(i,j \in [1,N])$, and $\boldsymbol{v}$ be a N-dimensional vector having components $v(i)$ $(i \in [1,N])$. Then $\boldsymbol{A}$ and $\boldsymbol{v}$ are decomposed into P submatrices and subvectors as

$$\boldsymbol{A} = \begin{pmatrix} \boldsymbol{A}_1 \\ \vdots \\ \boldsymbol{A}_P \end{pmatrix} \qquad \boldsymbol{v} = \begin{pmatrix} \boldsymbol{v}_1 \\ \vdots \\ \boldsymbol{v}_P \end{pmatrix} \tag{1}$$

so that each submatrix $\boldsymbol{A}_p$ having N_p rows and N_p-dimensional subvector v_p are assigned to the process $p \in [1,P]$ (see Fig. 1).

A process p performs a part of the matrix-vector multiplication $\boldsymbol{w} = \boldsymbol{A}\boldsymbol{v}$, namely $\boldsymbol{w}_p = \boldsymbol{A}_p\boldsymbol{v}$ to produce the N_p-dimensional subvector $\boldsymbol{w}_p$ of the product vector $\boldsymbol{w}$. For this operation, the process p needs a set of components of $\boldsymbol{v}$ namely $V_p = \{v(i) \,|\, a(i,j) \neq 0,\ K_p < i \leq K_{p+1},\ 1 \leq j \leq N\}$ where $K_p = \sum_{q=0}^{p-1} N_p$. The component set V_p is divided into disjunct subsets $V_p^1, \ldots, V_p^P$ such that $V_p^q = \{v(i) \in V_p \,|\, K_q < i \leq K_{q+1}\}$ to form the set of components to be sent from the process q to p.

Now let us concentrate on the sender/receiver process pair q and p to simplify the notations and let $V = V_p^q$. Let I be the set of indices of the components in V, i.e., $I = \{i \,|\, v(i) \in V\}$. The ascendingly ordered sequence $i_1, \ldots, i_k$ $(k = |I|)$ of indices in I is divided into disjunct *index fragments* $\phi(1), \ldots, \phi(m)$ such that $I = \bigcup_{j=1}^{m} \phi(j)$ and each of them has contiguous indices. That is, each $\phi(j)$ satisfies the followings where $h(j) = \min \phi(j)$ and $s(j) = |\phi(j)|$.

$$\forall i \in \phi(j):\ h(j) \leq i < h(j) + s(j) \tag{2}$$

$$\forall j \in [1, m-1]:\ h(j) + s(j) < h(j+1) \tag{3}$$

Then, with each index fragment $\phi(j)$, we define the *fragment* of vector components $f(j) = \{v(i) \,|\, i \in \phi(j)\}$.

Vector components in a contiguos sequence of fragments, or *chunk*, from $f(j)$ to $f(k-1)$, i.e., $F(j,k) = \bigcup_{l=j}^{k-1} f(l)$ can be transferred from the process q to p as a single message by one of the following ways. One way is to *pack* them into a message buffer of $\sum_{l=j}^{k-1} s(l)$ components and send the contents of the buffer minimizing the message size while paying the cost of copying. The other way is to *combine* the fragments without copying to have a message including not only them but *gaps* between them, making message size $h(k-1) + s(k-1) - h(j)$ larger than required.

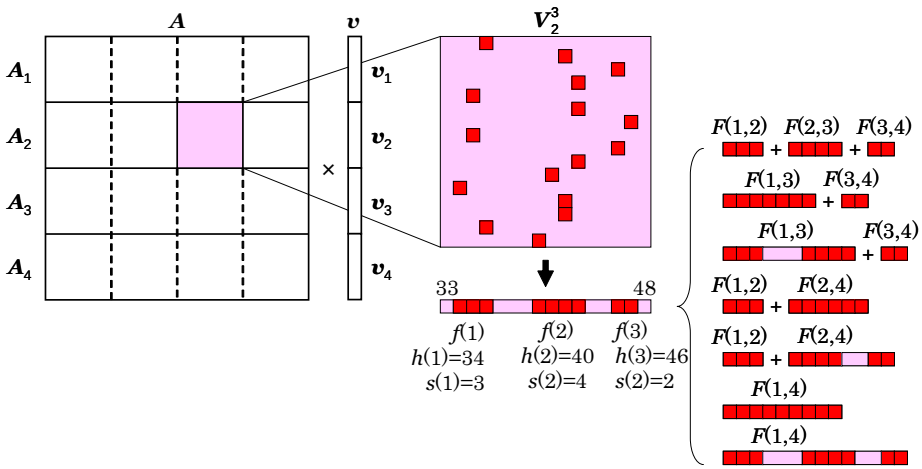


Fig. 1 Example of $\boldsymbol{A} \times \boldsymbol{v}$, fragments, chunks and assortments

The *cost* to transfer a fragment or a chunk is defined as follows. Let $C_T(n)$ be the non-decreasing function of n to give the time to transfer a message having n vector components from q to p, and $C_C(n)$ be that to give the time to copy n contiguous components from the vector store to the buffer for packing. With these cost functions, we can define the costs of the packing and combining transfer of a chunk $F(j,k)$ from q to p, namely $C_{pack}(j,k)$ and $C_{comb}(j,k)$ as follows.

$$C_{pack}(j,k) = C_T\left(\sum_{i=j}^{k-1} s(i)\right) + \sum_{i=j}^{k-1} C_C(s(i)) \tag{4}$$

$$C_{comb}(j,k) = C_T(h(k-1) + s(k-1) - h(j)) \tag{5}$$

Note that the second term of the equation (5) above is the sum of copying costs of the fragments rather than the cost for the sum of fragment sizes, because we have to take care of the overhead incurred in each fragment copy.

Then $C(j,k) = \min(C_{pack}(j,k), C_{comb}(j,k))$ gives us the cost to send $F(j,k)$ as a single message. Note that $C(j,j+1) = C_{comb}(j,j+1) = C_T(s(j))$ is the transfer cost of the single fragment $f(j)$, and their sum over $F(j,k)$, i.e., $\sum_{i=j}^{k-1} C(i,i+1)$ can be smaller than $C(j,k)$. In general, the minimum transfer cost for a fragment sequence is given by a series of packing/combining transfers of its subsequences, i.e., chunks.

Now we define our problem as follows: from a given fragments $f(1), \ldots, f(m)$, find the set of chunks $F(j_1,j_2), \ldots, F(j_n,j_{n+1})$, or the *assortment* of the fragments, where $j_1 = 1$, $j_{n+1} = m+1$, and $j_i < j_{i+1}$ for all $i \in [1,n]$, such that $\sum_{i=1}^{n} C(j_i,j_{i+1})$ is minimized.

3. Algorithm

To solve the minimization problem defined in the previous section, we introduce a subproblem defined as follows.

$$C_{\min}(n,k) = \min_{j_2,\ldots,j_n} \left\{ \sum_{i=1}^{n} C(j_i,j_{i+1}) \,\middle|\, j_1 = 1,\ j_{n+1} = k+1,\ j_i \le j_{i+1} \right\} \quad (6)$$

The cost $C_{\min}(n,k)$ above means the minimum transfer cost for fragments $f(1), \ldots, f(k)$ with at most n chunks which are $F(j_i,j_{i+1})$ with all j_i such that $i \in [1,n]$ to give the minimum but excluding those $j_i = j_{i+1}$ because $F(j_i,j_i)$ means a empty chunk by definition and thus $C(j_i,j_i) = 0$. It is clear that $C_{\min}(1,k) = C(1,k)$ from the definition above, and also that $C_{\min}(m,m)$ gives the minimum cost for the original problem.

Since $C(j_i,j_{i+1})$ can be calculated from any other $C(j_{i'},j_{i'+1})$, the definition above can be rewritten with the formulation similar to that presented in Ref. 1) as follows.

$$C_{\min}(n,k) = \min_{1 \le j \le k} \{C_{\min}(n-1,j) + C(j,k)\} \quad (7)$$

From the recurrence above and the obvious facts that $C_{\min}(n,k) = C_{\min}(k,k)$ for any $n > k$ and $C(j,j) = 0$ for any j by definition, we have the following code based on dynamic programming technique.

```
for k = 1 to m do  C_min(1,k) ← C(1,k);
for n = 2 to m do begin
  for k = n to m do begin
    c_min ← C_min(n−1,k);  j_min ← k;
    for j = k−1 downto n−1 do begin
      c = C_min(n−1,j) + C(j,k);
      if c < c_min then begin
        c_min ← c;  j_min ← j;
      end
    end
    C_min(n,k) ← c_min;  J_min(n,k) ← j_min
  end
end
```

The code above give us the optimum assortment of fragments by a sequence of their indices $j_2, \ldots, j_m$ which minimize the right-hand side of the equation (6) for $C_{\min}(m,m)$, i.e., the arg min of the equation, through a back-trace of $J_{\min}(n,k)$ as follows.

$$j_m = J_{\min}(m,m) \qquad j_n = J_{\min}(n,j_{n+1}) \quad (2 \le n < m) \quad (8)$$

It is clear that the code above takes $O(m^3)$ time providing that we can calculate $C(j,k)$ in the most-inner loop in $O(1)$ time for each. From equations (4) and (5), an $O(1)$ calculation is easily implemented by keeping track $\sum_{i=j}^{k-1} s(i)$ and $\sum_{i=j}^{k-1} C_C(s(i))$. As for the spatial complexity, it is also obvious we need $O(m^2)$ space to keep $J_{\min}(n,k)$[⋆1].

4. Experiments

4.1 Implementation and Evaluation Environment

We implemented the algorithm discussed in Section 3 together with the vector transfer for multiplying it to a sparse matrix represented in CRS format, using

⋆1 It is unnecessary to have $C_{\min}(n,k)$ for all $n \in [1,m]$ because only $C_{\min}(n-1,k)$ is required to calculate $C_{\min}(n,k)$, and thus the space for it is $O(m)$. The $O(m^2)$ for $J_{\min}(n,k)$, on the other hand, is essential.

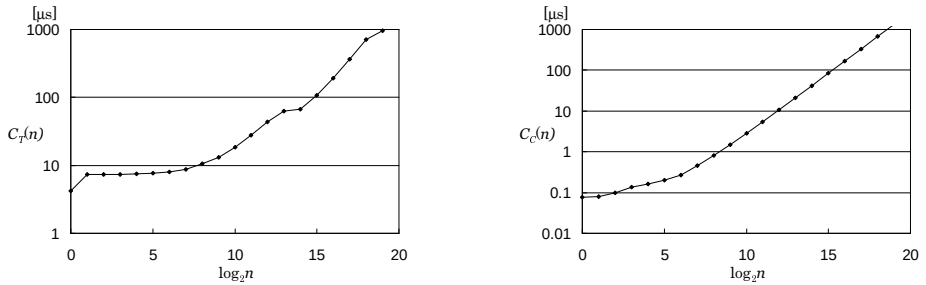


Fig. 2 Cost Functions $C_T(n)$ and $C_C(n)$.

C99 and MPI-2.0. The program consists of the following two procedures.

- A *scheduler* in which each process p analyzes the given submatrix $\boldsymbol{A}_p$ to find the optimum assortments for all $\boldsymbol{v}_q$ which the process requires for the multiplication with $\boldsymbol{A}_p$. Then all processes perform an all-to-all type communication to exchange the assortments so that each process p has assortments of $\boldsymbol{v}_p$ to send its components to other processes. In addition, the submatrix $\boldsymbol{A}_p$ is transformed into $\boldsymbol{A}'_p$ so that its entry $a'(i, j')$ has the entry $a(i, j)$ where the vector component $v(j)$ is j'-th one in the concatenation of all assortments for p. Thus the concatenation, namely $\boldsymbol{v}'_p$, satisfies $\boldsymbol{w}_p = \boldsymbol{A}_p\boldsymbol{v} = \boldsymbol{A}'_p\boldsymbol{v}'_p$. This procedure is executed just once at the beginning of, for example, a iterative method.
- A *transporter* in which each process p sends assortments of the given subvector $\boldsymbol{v_p}$ in non-blocking manner for combined chunks while packed chunks are sent in blocking manner to minimize the size of buffer for packing by sharing one buffer for all packed transfer. At the same time, the process p receives assortments in non-blocking manner to form $\boldsymbol{v}'_p$ together with the components of $\boldsymbol{v}_p$ itself which are copied locally. This procedure is executed repeatedly in iterative matrix-vector multiplications.

The scheduler is designed so that it accepts user-defined and environment-dependent cost functions $C_T(n)$ and $C_C(n)$. In our experiment we gave the scheduler simple table-driven functions based on the measurements of the transfer and copy performance of our evaluation environment, T2K Open Supercomputer[2] of Fujitsu's HX600 nodes comprising four quad-core Opteron 8356 for each. That is, we ran simple measurement programs using Fujitsu's C compiler version 3.0 and MPI library version 3.0, which are also used for the main program and its evaluation, to obtain the performance data shown in Fig. 2 for $n = 2^i$ vector components where $i \in [0, 19]$. Then each cost function with n components looks the table of performance data by $l = \lfloor \log_2 n \rfloor$ and $l+1$ to have the linearly interpolated value for n by;

$$C_x(n) \approx \begin{cases} \dfrac{(2^{l+1} - n)C_x(2^l) + (n - 2^l)C_x(2^{l+1})}{2^l} & n \le 2^{19} \\ \dfrac{n}{2^{19}} C_x(2^{19}) & n > 2^{19} \end{cases}$$

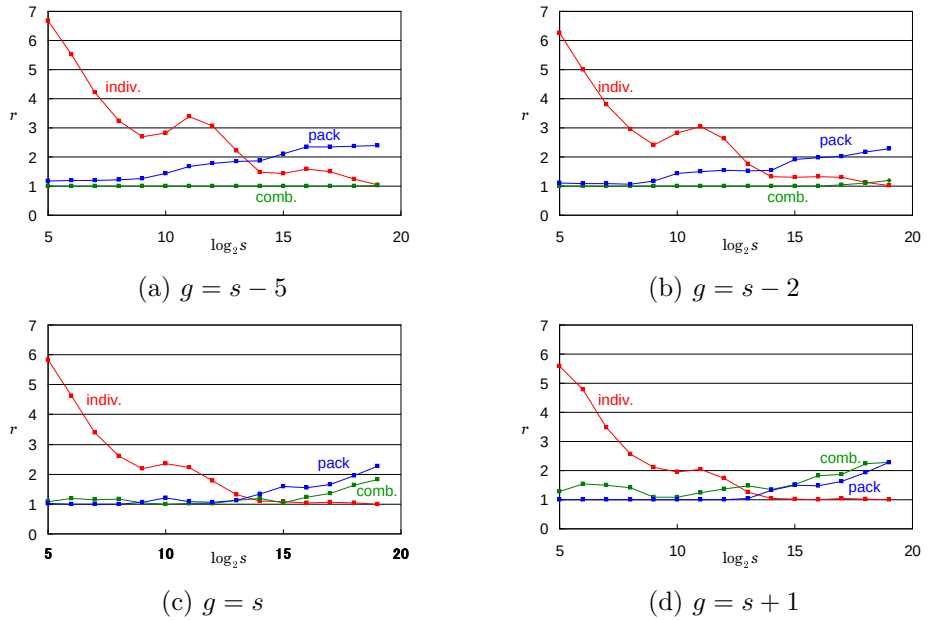
for $C_x(n) \in \{C_T(n), C_C(n)\}$.

4.2 Cost Optimality

The first experiment is to evaluate how small cost the optimum assortment gives compared to other simple methods. That is, we evaluated the costs of transferring ten fragments of various sizes and gaps between them with the following four methods; *individual*, *packing* and *combining* methods which sends all fragments individually, packing them into one message, and combining them and gaps to form one message, while *optimum* is our proposed one. We evaluated the amount of the cost using the cost functions rather than measuring the time for vector transfers. Therefore, the evaluation gives how our optimum method works well theoretically.

Fig. 3 shows normalized costs r relative to the optimum ones for randomly generated fragments having 2^s vector components on average, and 2^g components in each gap on average. Graphs in the figure are for the cases of $g \in \{s{-}5, s{-}2, s, s{+}1\}$, while s is varied from 5 to 19 in each case. Each value shown in the graphs is the average of ten experiments with specific settings of s and g.

It is observed from the graphs that the individual method incurs high costs when the size of fragments and gaps is small but becomes optimum when the size increases as easily expected. As for other two simple methods to send all fragments by one message, one of them is almost optimum for small size fragments but it depends on the gap size which method is better and (nearly) optimum. On the other hand, the optimum method gives not only the best of three simple



(a) $g = s - 5$ (b) $g = s - 2$

(c) $g = s$ (d) $g = s + 1$

Fig. 3 Costs of Simple Methods Relative to Optimum Method.

methods but also successfully finds optimum assortment of these three methods even with merely ten fragments. For example, in a case of $s = g = 13$, the optimum method finds three chunks of 4, 3 and 3 fragments to achieve about 10 % cost reduction from the best of three simple methods.

4.3 Vector Transfer Performance

The next experiment is to evaluate the real performance of vector transfers for large scale matrix-vector multiplications. We used 64 CPU cores equipped in four nodes of HX600 to allocate 64 MPI processes of a multi-processed matrix-vector multiplication.

Matrices are artificially generated using a *base* matrix derived from a difference equations using 7-point stencil for a 3-dimensional 256^3 grid space with periodic boundary. Since the base matrix $\boldsymbol{A}$ is constructed by a lexicographical ordering on grid coordinates, the $i-th$ row of the matrix has non-zero entries in its columns of $j_{i,1} = i$, $j_{i,2} = i \oplus 1$, $j_{i,3} = i \ominus 1$, $j_{i,4} = i \oplus 256$, $j_{i,5} \ominus 256$, $j_{i,6} = i \oplus 256^2$ and $j_{i,7} = i \ominus 256^2$ where $x \oplus y = ((x + y - 1) \bmod 256^3) + 1$ and $x \ominus y = ((x - y - 1) \bmod 256^3) + 1$.

To have matrices having some random sparseness, we transformed the base matrix $\boldsymbol{A}$ by shifting its non-diagonal elements using normal random numbers with mean 0 and a certain variance σ^2. That is, we have a matrix $\boldsymbol{A}^{(\tau)}$ whose i-th row has seven non-zero entries at $j_{i,1}$ and $j_{i,k} \oplus x_{6(i-1)+(k-1)}$ for $k > 1$ where x_l is the l-th element in a random number sequence $x_1, \ldots, x_{6 \cdot 256^3}$ with $\sigma = 2^{\tau \star 1}$.

A generated matrix of $256^3 \times 256^3$ or $2^{24} \times 2^{24}$ is then split into 64 submatrices of equal size, 2^{18} rows for each, to assign each of them to each process. Therefore, the vector $\boldsymbol{v}$ is 2^{24}-dimensional while each subvector $\boldsymbol{v}_p$ is 2^{18}-dimensional. Finally, in order to compensate excessive randomness and sparseness of the matrix resulting in a unrealistically large number of fragments in V_p^q for a process pair p and q due to our straightforward generation with random shifting, we reduce the number of fragments to 2^{10} by filling small size gaps with virtual components.

We measured the time of vector transfer varying σ from 0 for $\boldsymbol{A}$ to 2^{24} for $\boldsymbol{A}^{(24)}$ which are almost uniformly random sparse. For each matrix, the times with the individual and packed method are also measured together with the optimum method. In addition, we measured the time of `MPI_Allgatherv()`, which is independent from σ, to give the whole of vector $\boldsymbol{v}$ to all processes.

Fig. 4 shows measured transfer times in (a) and the ratio of those of the packing method over the optimum method in (b). Note that $\tau = 0$ in Fig. 4(a) does not mean $\sigma = 2^0 = 1$ but means $\sigma = 0$. The graphs exhibit that the packing and optimum methods work well taking about 10 ms or less and almost tie for $\tau \leq 16$, which means vector components in a subvector $\boldsymbol{v}_p$ are required only by two adjacent processes $p \pm 1$ almost always. In this range, they are much faster than the individual method, which is also faster than all-gather.

Then the transfer times of the optimum and packing methods start growing but still tie until $\tau = 19$ at which about 10 processes needs the components in a subvector, while that of the individual method steeply grows due to a flood of small size messages. After that the optimum method clearly shows its advantage

⋆1 Therefore, the matrix is meaningless in the sense of mathematics nor physics.

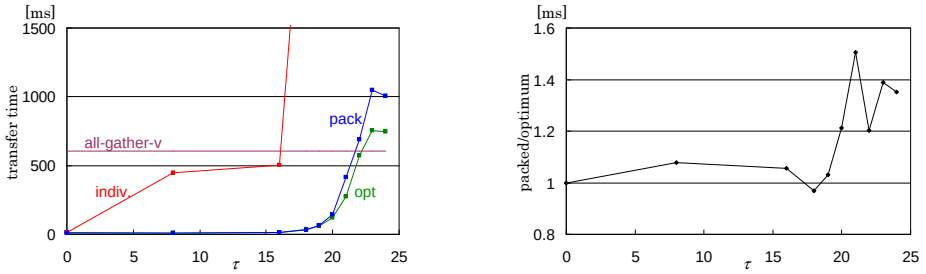


Fig. 4 Vector Transfer Time (a) and Comparison of Optimum and Packing Methods (b).

over the packing with 20–50 % performance superiority. However, their advantage over all-gather disappears when τ is large to make it necessary for a process to gather fragments from all other processes, due to our simple communication scheduling with a batch of non-blocking receives followed by a series of non-blocking (for optimum) or blocking (for packing) sends.

5. Conclusion

In this paper, we discussed an efficient algorithm to exchange fragments of a block-decomposed vector among processes for iterative sparse matrix-vector multiplication with a matrix also block-decomposed. For a process pair p and q and the transfer of the fragments in the subvector residing in q and required by p for its local multiplication, our algorithm analyze p's submatrix to find the optimum assortment of the fragments for individual, packing or combining transfer of them using dynamic programming technique.

We implemented the algorithm on our T2K Open Supercomputer and measured its performance using 64 CPU cores to which 64 MPI processes are allocated. The evaluation with artificially generated matrices of $2^{24} \times 2^{24}$ confirms our optimum method is much faster than all-gather type method when the fragments are not widely distributed among processes, and outperforms the packing method especially when the size of gaps between fragments varies widely.

Our urgent future work is to evaluate our method with realistic matrices. We also plan to improve the performance of our method by the followings; introduce some heuristics to reduce $O(m^3)$ time complexity in a practical sense; use more accurate cost functions to incorporate the effect of process mapping, memory layout of fragments, and so on; improve the communication scheduling especially when communication pattern is (almost) all-to-all.

Acknowledgments A part of this research work is pursued as a Grant-in-Aid Scientific Research #20300011 and #21013025 supported by the MEXT Japan.

References

1) Konishi, M., Nakada, T., Tsumura, T., Nakashima, H. and Takada, H.: An Efficient Analysis of Worst Case Flush Timings for Branch Predictors, *IPSJ Trans. Advanced Computing Systems*, Vol.48, No.SIG8 (ACS18), pp.127–140 (2007).
2) 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).