IPSJ Transactions on Programming Vol.5 No.1 17–25 (Mar. 2012)

Regular Paper

Design and Implementation of a High Productivity Language with Communication Aggregation for Parallel Scientific Computation

Katsuaki Ikegami[1,a)] Kenjiro Taura[1]

Received: June 27, 2011, Accepted: November 8, 2011

Abstract: Message passing model is a popular programming model in which users explicitly write both "send" and "receive" commands in programs and locate shared data on each process's local memory address. Consequently, it is difficult to write a program with complicated algorithms using a message passing model. This problem can be solved using a partitioned global address space (PGAS) model, which can provide virtual global address space and users can effortlessly write programs with complex data sharing. The PGAS model hides the network communication as implicit global memory access. This leads to better programmability, but there can be additional network communication overhead compared to a message passing model. We can reduce the overhead if programs read or write global memory in bulk, but this complicates writing programs. This paper presents the programming language and its runtime to achieve both the programmability and the performance with automatic communication aggregation. The programmer can write global memory accesses in the normal memory access style; then, the compiler and runtime aggregate the communication. In particular, the most time-consuming network accesses are placed in loops, and therefore, this paper suggests how the compiler detects global memory accesses in loops and aggregates them and how to implement that idea.

Keywords: PGAS, parallel programming, communication optimization

1. Introduction

1.1 Background

Scientific computing is one of the main applications of current parallel computing systems. Typical large-scale scientific applications include earthquake, weather, and fluid dynamics simulations. These applications are computed using numerical algorithms, for example, the finite element method or the particle method. When these simulations are executed in parallel, problems are usually divided into multiple areas that are assigned to processors. For example, in the finite element method, the problem domain is partitioned into small elements and connections between elements, and each process handles some elements. Connections between elements are static, but they are not always as simple as a grid, and therefore, element access patterns for computing an element force can be irregular. In the particle method, we divide the problem area into many small particles and compute forces between particles. Since connections between particles are dynamic, access patterns during computation are not only irregular but also dynamic.

One of the most popular libraries for parallel programming is the Message Passing Interface (MPI), which is based on the message passing programming model. Data are stored in local memory, which is only readable by the owner processor. When data need to be shared among processes, users have to explicitly write message send and receive API calls in both sender and receiver processes. Furthermore, a sender must specify the data to be sent and the receiver, and a receiver must specify the buffer to store received data and the sender of the data. In this complicated programming model, a programmer must handle all the pair of senders and receivers explicitly, which is a difficult task in dynamic or irregular computation. The shared memory programming model is different from this programming model. For instance, users only have to access memory space where data are shared in the shared programming model.

Accordingly, writing programs with the message passing model is much harder than that with the shared memory programming model. However, many parallel systems, such as a cluster, are distributed memory systems that do not provide a shared memory interface. This is the main reason why the message passing model is mostly used. To improve programmability, the partitioned global address space (PGAS) [1] programming model has been studied. The PGAS model provides a shared memory interface, and the message communications between processes are hidden by runtime. Automatic parallelization of sequential programs or completely transparent distributed parallel shared memory has been previously studied, but the PGAS model is characterized by data locality. In the PGAS model, users explicitly specify the place (node or process) of data. Thus, using the PGAS model makes it easy to implement algorithms because of the shared memory interface, and high performance with explicit data affinity can also be achieved.

[1] The University of Tokyo, Bunkyo, Tokyo 113–8656, Japan
[a)] liquid@logos.ic.i.u-tokyo.ac.jp

However, current implementations of PGAS model languages suffer from a trade-off between programmability and performance. When a program accesses the data in the global shared address space, the process that performs an access and the data owner process need to communicate with each other. In a distributed system, processes on different nodes cannot communicate through memory but must do so through the network. However, network latency is much higher than that of memory, and thus, frequent access to the global address space over the network can cause significant overhead and performance degradation. Therefore, the global address space should be accessed in bulk to achieve better performance. This fact leads us to write a program with the following characteristics: the program reads the necessary data from the global address space, computes using local memory, and then writes back the results to the global address space. In this programming style, users have to manage which local buffer address corresponds to the original global address; this becomes a nontrivial problem when the access pattern to the global address space is irregular, in which case users cannot easily write programs even with PGAS.

1.2 Objective

In this paper, we propose a programming language based on the PGAS model and its runtime that provides high productivity with complete global address access even in a distributed system, as well as achieving high performance comparable to existing runtime based on the message passing interface model. Using our language, the runtime aggregates accesses to the global address space into bulk accesses, and thus, programmers can write access to global address space in a consistent style.

2. Related Work

In this section, we compare the productivity of implementing an irregular application among several existing programming models. This application computes the displacement of a body subjected to an external force and approximates the body as nodes and static connections between nodes as shown in **Fig. 1**. The circles in Fig. 1 are the nodes, and the lines between circles are the connections. We assume that the inner force acts only on the connections. Each step of computation consists of a computation of forces of each connection and a computation of displacements of each element. The real-world applications also divide the target problem area into small pieces and compute interaction forces between elements and their displacements in each time step, so this simple application is a good example of real-world scientific applications. We assume that the problem is distributed on three processes and clarify what program description is needed in #2 processor. In Fig. 1, the area written as #1 is the first process's area. Additionally, problem division is done in runtime, so static analysis of the access pattern in compile time is not possible.

2.1 Message Passing Model

Message passing model languages provide send and receive communication between processes on distributed systems. They also provide typical group communication. The message passing model is the lowest level programming model in distributed parallel programming. If we assume that the problem is computed using the message passing model, the second process would communicate as shown in **Fig. 2**. This figure shows that the first process will send the zeroth, first, third, and fourth elements of data in its local buffer, and the third process will do the first, third, and fourth. The second process will place received data from the fourth to tenth of its local buffer. In this operation, nodes are never accessed by the global index over nodes but the index local in the process. The processes should handle the connections between nodes with the local buffer index. This example illustrates the difficulty in implementing irregular computation with the message passing model even for this simple application.

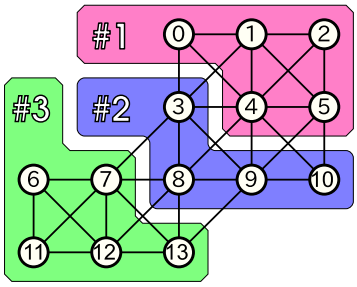


Fig. 1 Irregular divided nodes.

2.2 Global View PGAS Model

Sharing data with process-oriented addresses makes programming hard. One solution is to provide global addresses; these can be used by all the processes on each node. The PGAS model can be used to realize this solution.

2.2.1 High-level Global View PGAS Model

First, we examine high-level PGAS languages, for example, Chapel [2] and X10 [3]. In those languages, accesses to data on global addresses are transparent as normal memory access, and thus, programmers do not have to be aware of whether or not the system is distributed. Chapel also provides a data-parallel programming model and an object oriented programming model. The second process in Fig. 1 can access the neighbor nodes as in **Fig. 3**. In Fig. 3, the arrows mean that to-side nodes use from-side nodes for the force computation. This figure shows that processes can access node data on any processes using global addresses.

However, current runtime implementations of these languages do not aggregate communication and thus do not achieve high performance.

2.2.2 Low-level Global View PGAS Model

Consider, now, lower level PGAS languages, for instance, Unified Parallel C (UPC) [4] or Distributed Memory Interface (DMI) [5]. The former is an extension of the C language, and the latter is the library providing the C language API. Both languages provide a local normal pointer and a pointer to the global address space. Shared data are accessed via the global pointer. With these low-level PGAS languages, users access the global address space by copying between local memory and the global address space explicitly. Using UPC, users can access the global pointer directly, but this involves frequent communications and is not usually used in critical codes. Instead, users should use `upc_memget` or `upc_memput` to access global address space.

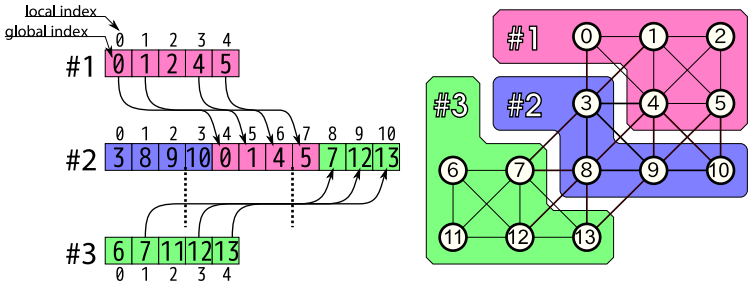


Fig. 2 Communication of node data with message passing model.

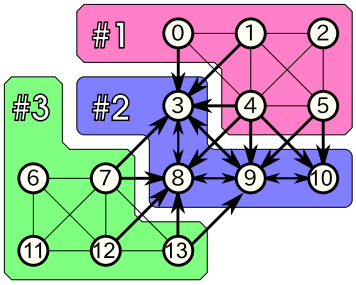


Fig. 3 Sharing node data with high-level PGAS languages.

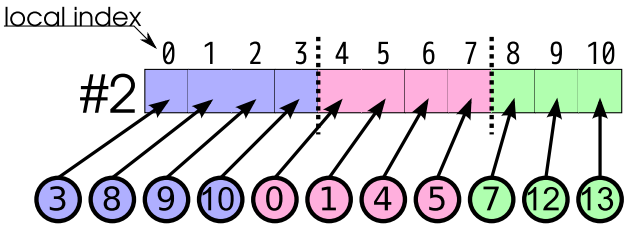


Fig. 4 Reading node data to local buffer with low-level PGAS languages.

When a user computes the problem of Fig. 1 with aggregation of communication with these languages, all the needed nodes should be loaded to the local buffer as shown in **Fig. 4** before actual computation begins.

This program description is easier than the message passing model because nodes are read using a global address. However, a programmer must handle the local buffer address during the calculation phase, which makes it difficult to implement irregular algorithms using low-level PGAS languages.

2.3 DMI

In this section, we describe the DMI, i.e., the PGAS model library. As shown above, the DMI is the C library providing the global address space. Users can manage data consistency within the global address space with any affinity, and also the DMI provides asynchronous read/write, read/write to/from discrete addresses, several basic synchronizations, and user-defined atomic operations, which are useful for parallel programming. Programs in the DMI can achieve performance as high as that of MPI programs.

Using the DMI, users cannot read/write transparently to the global address space, but users can copy between global address space and local memory, as shown in **Fig. 5**. A useful feature of the DMI compared to UPC is that the DMI can access the global address without specifying which address is on which node, whereas UPC API can access the global address only on one node.

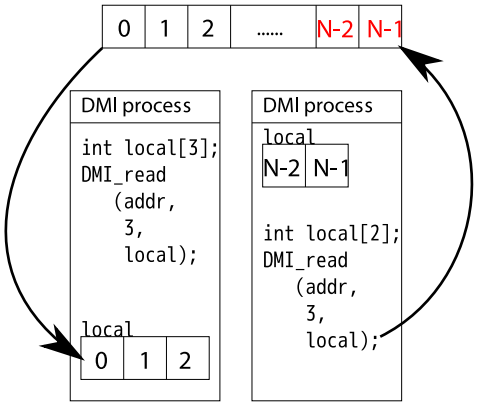


Fig. 5 Basic concept of DMI.

2.4 PGAS Runtime Optimizations of Irregular Access

Existing PGAS languages have tried to resolve the trade-off between ease of programming and performance. In the case of UPC, Wei-Yu Chen optimized communication by communication overlap and aggregation of communication [6].

Communication overlap occurs when the runtime begins to read global address space earlier than the data are actually needed and this delays the completion of writing the data. The readable points or the delayable write completion points are obtained through a static analysis of the compiler. This technique is effective for a large amount of read or write operations.

The goal of aggregation of communication is to aggregate communications of global accesses and reduce communication overhead. Upon implementation, an aggregated communication sends/receives all data covering necessary data; hence, it cannot aggregate accesses on sparse data efficiently.

Another PGAS language, Titanium, also optimizes communications with an inspector/executor strategy [7], [8], in which something is prepared before real computation (*inspector*) and is used in later computation (*executor*.) If the executor phases run much more than inspector phases, the cost to prepare is reduced. The compiler should distinguish which operation is done with a single inspector. In the Titanium specification, for an array on global address space `A` and its indices array `B`, the compiler uses this strategy if the codes repeatedly access `A[B[i]]` while `A` and `B` are unchanged. In the inspector phase, all the global addresses for `A[B[i]]` for all i are collected; the local received data are then used in the executor phase. Titanium takes the best attributes from various types of aggregation: reading exact needed

data, reading the data covering the needed data, and reading all the array. However, Titanium only aggregates read access, not write access.

Lastly, high performance Fortran (HPF) [9] also provides global address space. With HPF, a programmer can parallelize a sequential Fortran program with some directives. HPF provides data-parallel or task-parallel syntax, and data distribution is automatically determined by runtime or explicitly specified by the programmer. In HPF, HALO [10] is the feature for communication aggregation. HALO directives specify which data each process accesses and use this information in runtime to communication optimization, for instance, aggregation or overlapping. The problem with HALO is that programmers must explicitly notify the access data in the program, so it is hard to write a program with a dynamic access pattern.

3. PGAS Language with Automatic Optimization of Communication

3.1 Basic Design

As shown above, existing PGAS model languages suffer from a trade-off between performance and programmability, and optimization techniques in compile time or runtime make it possible to achieve high performance with simple programming. Thus, the programming language should allow all the global accesses with normal memory access syntax, and the runtime should optimize the communication. In this paper, we propose the high-level PGAS language and its runtime with almost transparent access to the global address space over the DMI. To implement this concept, we use an extension of C++, translate the source codes to normal C++ code using the DMI with a translator script, and compile into an execution file with the normal C++ compiler GNU Compiler Collection (GCC) [11].

3.2 Programming Model

This language is based on the thread programming model of the DMI. Users describe thread creations and joins in the `DMI_main` function, which is a predefined DMI function same as `main` in normal programs. Users can also use the SPMD programming model provided by the DMI. That means that this language does not affect the whole programming model; thus, programs of the language are not parallelized implicitly.

3.3 C++ Extension

The suggested language extends C++, by adding two syntaxes: `shared` type specifiers and a `parallel_for` statement. The `shared` type specifier distinguishes global address space from local memory space. The `shared` type specifier has the following properties.

- To assign a `shared` expression into a not `shared` expression means reading data from the global address space.
- To assign a not `shared` expression into a `shared` expression means writing data from the global address space.

For instance, the expression `sum = a[0]` with the pointer to shared data `a` and normal variable `sum` means that the value on the global address space pointed `a` assigned to `sum`. Another expression `a[0] = 8` means writing to global address space. We can write a program to accumulate `a[0]+a[1]+...+a[N-1]` with `shared` integer pointer `a`, as shown in **Fig. 6**. In fact, this program communicates on each access `a[i]` and suffers from too much communication overhead. In the next subsection, the `parallel_for` statement is proposed as the solution.

```
1  shared int * a;
2  int sum=0;
3  for(int i=0; i<N; ++i)
4    sum += a[i];
```

Fig. 6 Example of a shared pointer: sum of array.

```
1  shared int *A, *B, *C;
2  int *p;
3  #ifdef USE_PARALLEL_FOR
4  parallel_for(int i=0; i<=N; ++i)
5  #else
6  for(int i=0; i<=N; ++i)
7  #endif
8  {
9    ...
10   A[i] = B[p[i]] + C[i+1];
11   ...
12 }
```

Fig. 7 Example of parallel_for: aggregation of global accesses with indirect reference.

3.4 parallel_for Statement

The `parallel_for` statement is almost equivalent to the normal `for` statement in C/C++. The difference is that iterations of `parallel_for` are not executed in order. It is confusing for users that the `parallel_for` statement is not a data-parallel syntax but aggregates communication using concurrency of iteration. All the iterations are done concurrently but serialized.

For example, assume that we write the program shown in **Fig. 7** using the `for` statement or the `parallel_for` statement. In the code, the current process handles the data in $[0, N]$. With the normal `for` statement, this program would execute as shown in **Fig. 8**. In the figure, blue means a global read and orange means a global write. Figure 8 shows that for each $i = 0, \ldots, N$ read on B and C, the result is computed and the value written to A. In this flow, the number of communications is $3N$, and the overhead is too large.

Meanwhile, iterations of this program do not depend on each other, and thus, the `for` can be replaced with a `parallel_for`. If a `parallel_for` is used, runtime can execute the program as shown in **Fig. 9**. First, runtime collects all the addresses of `A[i]`, `B[p[i]]`, and `C[i+1]` for $i \in [0, N]$. This is represented by the green color in Fig. 9. Second, all the data in bulk are read for the collected addresses of `C[i+1]` and `B[p[i]]`. After bulk reading is completed, a real calculation is done for each `i` in local memory. Finally, all the data are written to the global address. In this execution flow, the number of communications is only three. The runtime can aggregate communication of global access in `parallel_for` as shown above.

3.4.1 Redundancy Elimination

In this section, we investigate a program such as that shown in **Fig. 10**. When this program is optimized as shown above, the program is executed until adding one to `a[i]`, and write the data to `a` and read `a` again before doubling the value. However, because

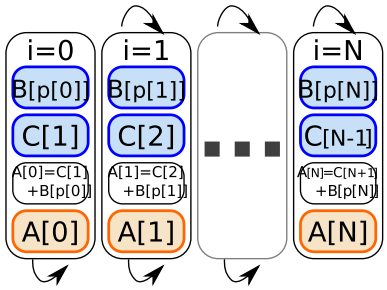


Fig. 8 Flow of normal for statement.

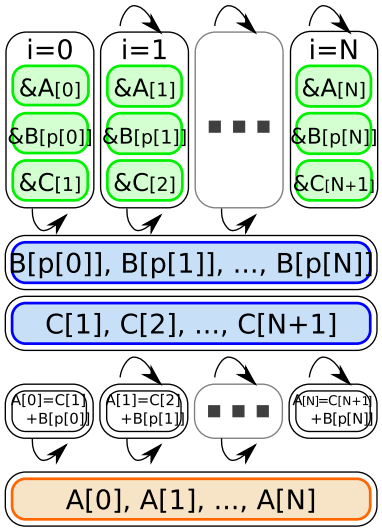


Fig. 9 Flow of parallel_for statement.

```
1 shared int * a;
2 parallel_for(int i=0; i<N; ++i)
3 {
4   a[i] += 1; // increment
5   a[i] *= 2; // doubling
6 }
```

Fig. 10 Example of parallel_for: access to the same address again.

the DMI uses sequential consistency, it assumes that a write to a global address is guaranteed to be read in a later operation only in the same process before synchronization. This consistency model allows the program to reduce first write and second read to `a` if there is no synchronization during adding one and doubling. Instead of doing a global read or write, the program reuses the local memory area.

3.4.2 Inspector/Executor

The suggested runtime should collect all the addresses to access first, sort the addresses, and then prepare some data for communication. This costs $O(N \log N)$ time complexity to save N addresses, which is too heavy to do in every `parallel_for` execution, so the runtime uses inspector/executor strategy when the access pattern is not changed during execution.

3.5 Implementation

In this section, we describe the implementation of compiler and runtime to meet the language specification. We implemented a translator from the language to C++. For instance, this translator translates the original source in Fig. 7 into the C++ code in **Fig. 11**. The `MyGroup<>` template class instance

```
1  int * a; // DMI pointer
2  int sum=0;
3  MyGroup<int> _G1;
4  for(int i=0; i<N; ++i)
5  {
6    _G1.pushPtr(a+i); //L1
7  }
8  _G1.init(); //L2
9  _G1.read(); //L3
10 for(int i=0; i<N; ++i)
11 {
12   int & _T1 = _G1.atAndNext(); //L4
13   sum += _T1; //L5
14 }
```

Fig. 11 Translated code of the example of parallel_for: sum of array.

```
1  int * a; // DMI pointer
2  MyGroup<int> _G1;
3  for(int i=0; i<N; ++i)
4  {
5    _G1.pushPtr(a+i); //L1
6  }
7  _G1.init(); //L2
8  _G1.read(); //L3
9  for(int i=0; i<N; ++i)
10 {
11   int & _T1 = _G1.atAndNext(); //L4
12   _T1 += 1; //L5
13 }
14 _G1.resetCounter(); //L6
15 for(int i=0; i<N; ++i)
16 {
17   int & _T1 = _G1.atAndNext(); //L7
18   _T1 *= 2; //L8
19 }
20 _G1.write(); //L9
```

Fig. 12 Translated code of the example of parallel_for: access to the same address again.

`MyGroup<int> _G1` manages the pointer collection, aggregated read and write, and local memory buffer. At point L1, we add all the addresses `a+i` to `_G1`. The current implementation can only handle the bracket syntax access to shared pointers such as `p[i]`. Next, runtime checks the address in `_G1` to determine whether it is continuous or sparse at L2. If it is sparse, runtime also initializes a DMI object for sparse access. Next, read the `a` data from global address space at point L3. At L4 and L5, the program accesses the local buffer in `_G1`. The member function `MyGroup<>::atAndNext()` at L4 is the function returning the memory area reference in address registered order.

For another example, the Fig. 10 program is translated into **Fig. 12**. In this program, `a[i]` of incrementing and `a[i]` of doubling should be compared by compiler data flow analysis to determine whether or not it is the same. In the current implementation, however, they are compared literally. Synchronization existence checking between two `a[i]` accesses depends on the existence of a function call in the current implementation. Synchronizations of the DMI is used as a function call, so this can check for synchronization. There is a problem that function calls without synchronization cause unnecessary reading and writing to `a`, which can add performance overhead. At L6 in Fig. 12, runtime resets the access counter of `_G1` instead of writing back `_G1` data and registering pointers and reading again to a new instance `_G2`.

4. Performance Analysis

4.1 Experiments

4.1.1 Experimental Environment

This analysis is executed on hosei and huscs clusters of the InTrigger platform [12]. **Table 1** shows further information.

4.1.2 Abstract of Experiment

In the experiment, sparse matrix vector multiplication (SpMV) and conjugate gradient (CG) are implemented as basic operations in scientific computation in the MPI, the DMI, and the presented language.

4.1.3 SpMV

SpMV is an algorithm with the simple code

$$A\vec{x} = \vec{b}. \quad (1)$$

Storing a sparse matrix in a two-dimensional array is memory inefficient, so a sparse matrix is usually stored in four arrays:

dias Diagonal components of the matrix are stored. The size is the same as that of the matrix.

vals Nondiagonal nonzero components of the matrix are stored.

cols The number of columns of each val item is stored.

rows The indices of each first-column component in vals are stored. The size is the same as the matrix size plus one.

The program code to compute SpMV with the following data structure is in **Fig. 13**.

To execute this program over a distributed system, vectors `x` and `b` should be distributed over all the processes. In particular, `x` should be readable from all the processes. By contrast, the matrix must not be shared because it does not change during execution. Thus, with the message passing model, first the vectors `x` and `b` are aligned to each process, and matrix `dias`, `vals`, `rows`, and `cols` are distributed to each processor. In the real execution phase, the data of other processes are needed in access `x[cols[j]]`. Using the message passing model language, the process scans positions of `x` that should be received from corresponding processes and interchanges the data and receives what data should be sent to other processes. When the program executes a real computation, the process sends and receives some of `x`. Thus the message passing model is too complicated to implement even for such a simple algorithm.

Our second check for implementing this algorithm uses a complete global PGAS. `x` and `b` are still shared over the global address space. If communications are not aggregated, all the accesses during the diagonal calculation in 2 line and the nondiagonal calculation in 4 line cause communication, so communication overhead must be huge. If the user codes to read `x[i]` and `x[cols[j]]` to local memory in bulk, overhead is efficiently reduced but the programmer suffers from handling correspondence between the original `x[i]` and `x[cols[j]]` and the local buffer.

In our proposed language, the program should be coded as shown in **Fig. 14** to aggregate communications. It is slightly confusing since the matrix is locally distributed, but the code can be rewritten Fig. 13 to handle from `low` row to `high-1` row. In this code, the `parallel_for` statement first collects all the pointers of `x[i]`, `b[i]`, and `x[cols[j]]` and then performs a computation using the local copy. Finally, the result `b` is written back to the global address space. This aggregation is implicit.

The sparse matrix used in our experiment is chosen as **Table 2** from the University of Florida Sparse Matrix Collection [13]. **Figure 15** shows the shape of this matrix.

4.1.4 CG Method

The CG method is an iterative method for solving linear equations with symmetric and positive-definite matrices. The detailed algorithm is shown in **Fig. 16** [14]. This is an excellent small real-world program using SpMV.

Table 1 Environment of experiments.

cluster	huscs
# nodes	19
CPU	Xeon E5530 (2.40 GHz)
# core	8
memory	24 GB
network	10 G Ethernet
kernel	Linux 2.6.26-2-amd64

```
1 for(i=0; i<N; ++i) {
2    b[i] = dias[i] * x[i];
3    for(j=rows[i]; j<rows[i+1]; ++j) {
4       b[i] += vals[j] * x[cols[j]];
5    }
6 }
```

Fig. 13 Sequential code of sparse matrix vector multiplication.

4.2 Results

4.2.1 Programmability

In this section, the programmability of using the MPI, the DMI, and our proposed language is compared by looking at the number of lines of code. Programmability is based not only on quantity but also on the direct handling ability of the global address space, but only the quantity of code is only used. **Table 3** shows the lines of code of the three language implementations of the program. The "notable section" is the codes calculating SpMV or CG method, not including initialization or matrix loading. This comparison shows that the proposed language has a higher productivity than the usual message passing model or the normal PGAS model.

4.2.2 Performance

The scalability of SpMV and CG with three matrices is shown in **Figs. 17**, **18**, **19**, **20**, **21**, and **22**. The speed of the MPI with one process is the baseline. We also attempted to evaluate the DMI version without aggregation; however, the method is too slow and cannot measure execution time.

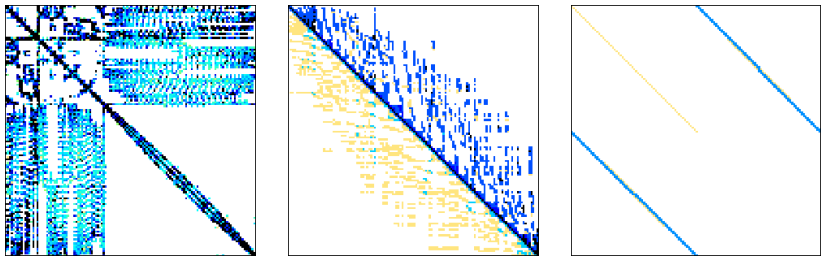
The result shows that the scalability of all the implementations breaks down for multinode systems. One reason for this breakdown is that the matrices are too small for the number of processes; even calculating the largest matrix nlpkkt200 SpMV costs

```
1 shared double *b, *x;
2 parallel_for(i=low; i<high; ++i) {
3    b[i] = dias[i-low] * x[i];
4    parallel_for(j=rows[i-low] - cols[0];
         j<rows[i+1-low] - cols[0]; ++j) {
5       b[i] += vals[j] * x[cols[j]];
6    }
7 }
```

Fig. 14 Implementation with the present language of sparse matrix vector multiplication.

Table 2 Matrices used in experiments.

id	Name	Cols	Number of nonzero nondiagonal components
1252	audikw_1	943,695	76,708,152
916	cage15	5,154,859	94,044,692
1904	nlpkkt200	316,240,000	431,985,632



audikw_1 cage15 nlpkkt200

Fig. 15 Image of matrices.

$\vec{r} \leftarrow \vec{b} - A\vec{x}$
$\vec{p} \leftarrow \vec{r}$
$norm \leftarrow |\vec{r}|$
for $k = 0, 1, \ldots$ **do**
  $\alpha = \dfrac{norm^2}{\vec{p}^T A\vec{p}}$
  $\vec{x} \leftarrow \vec{x} + \alpha\vec{p}$
  $\vec{r} \leftarrow \vec{r} - \alpha A\vec{p}$
  $norm' \leftarrow |\vec{r}|$
  if $norm' < \varepsilon$ **then**
    break
  end if
  $\beta \leftarrow \dfrac{norm'^2}{norm^2}$
  $\vec{p} \leftarrow \beta\vec{p} + \vec{r}$
  $norm \leftarrow norm'$
end for
return $\vec{x}$

Fig. 16 CG algorithm.

Table 3 The numbers of lines of applications with several implimentation.

		MPI	DMI	Proposed language
SpMV	whole codes	294	286	207
	notable section	121	86	24
CG	whole codes	346	339	224
	notable section	174	140	54

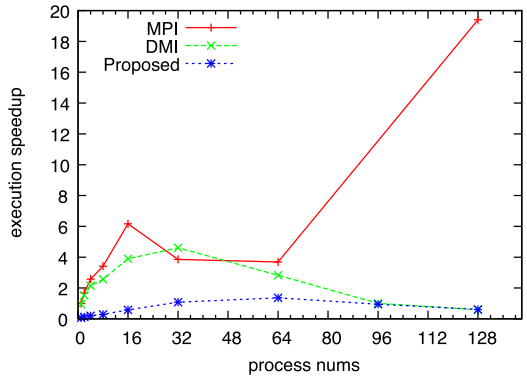


Fig. 17 SpMV, audikw_1.

only 0.1 s with one iteration with 128 processes. This means that overhead is dominant in the execution time.

In the performance graphs shown, the initialization cost referred to in Section 3.4.2 is not included. This is because the proposed language has a cost only in the first iteration, and the

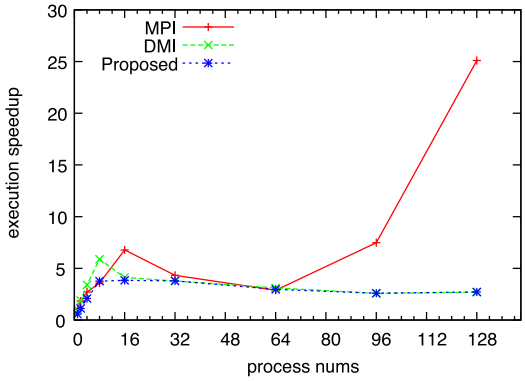


Fig. 18 CG, audikw_1.

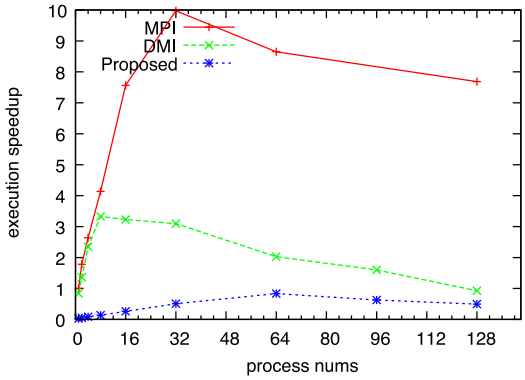


Fig. 19 SpMV, cage15.

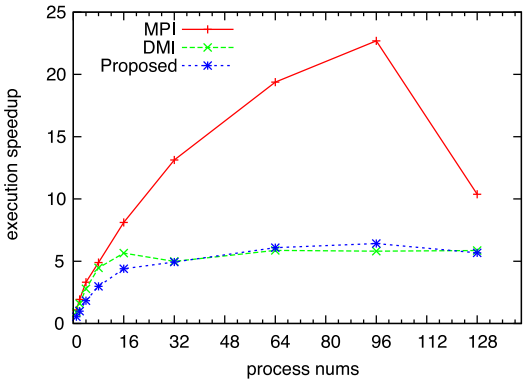


Fig. 20 CG, cage15.

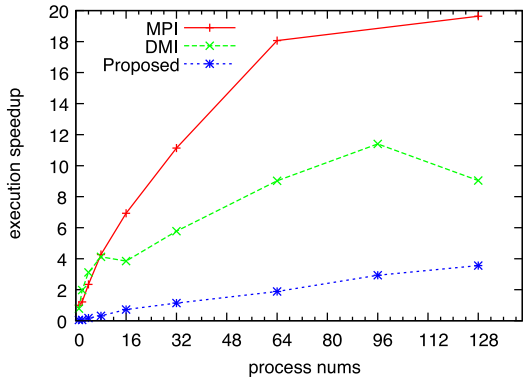


Fig. 21 SpMV, nlpkkt200.

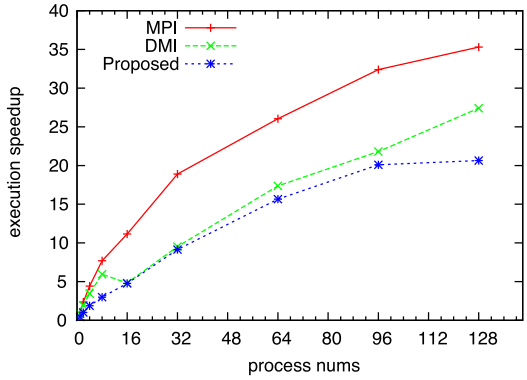


Fig. 22 CG, nlpkkt200.

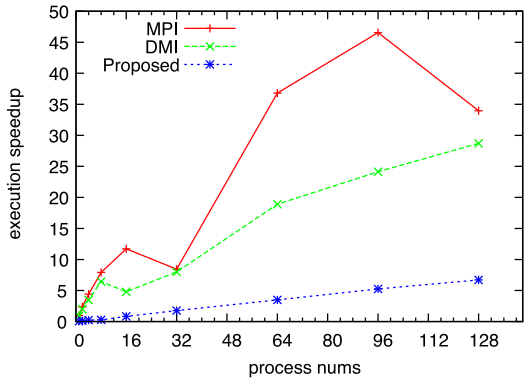


Fig. 23 CG, nlpkkt200, first iteration.

MPI and the DMI also have costs associated with aggregate communication. **Figure 23** shows the scalability with initialization cost in the presented language. This graph shows that initialization of aggregation information degrades performance and it is important to reduce initialization if the communication pattern does not change.

5. Conclusion and Future Work

The message passing model is the most widely used programming model in distributed parallel programming, but a programmer must write send and receive explicitly to share some data and must keep in mind which node the data are on. The PGAS programming model provides a global address space and a programmer can program with shared memory semantics. However, if access affinity to the global address space is small, communication overhead begins to dominate, and so a programmer should read the entire set of required data and compute on local memory. This presentation suggests that the PGAS language can help balance transparent access to global address space and performance with communication aggregation. Compared to existing libraries MPI and DMI, we have shown that our proposed language has much higher productivity, but its performance is about 57% of that of MPI.

The proposed language only provides `shared` and `parallel_for` syntax and leaves basic operation to the DMI, but these operations also should be wrapped with the language. The current translator does not analyze program semantics, so it is not checked whether or not the `shared` specifier is correctly propagated. It also does not analyze data flow, and thus, it is not checked which expression points to the same address as another expression. Data flow analysis also makes it possible to overlap communication. The inspector/executor strategy used in the performance experiment depends on data flow analysis; an automatic inspector/executor needs a much richer compiler analysis.

For measure performance, SpMV is too easy to implement, and more complicated applications are better to evaluate language productivity. For instance, molecular dynamics is better since the communication pattern will change depending on the movement of molecules.

Reference

[1] Yelick, K., Bonachea, D., Chen, W.-Y., Colella, P., Datta, K., Duell, J., Graham, S.L., Hargrove, P., Hilfinger, P., Husbands, P., Iancu, C., Kamil, A., Nishtala, R., Su, J., Welcome, M. and Wen, T.: Productivity and performance using partitioned global address space languages, *PASCO '07 Proc. 2007 International Workshop on Parallel Symbolic Computation* (2007).

[2] Callahan, D., Chamberlain, B.L. and Zima, H.P.: The Cascade High Producivity Language, *9th International Workshop on High-Level Parallel Programming Models and Supportive Environments* (*HIPS 2004*), pp.52–60, IEEE Computer Society (2004).

[3] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C. and Sarkar, V.: X10: An Object-oriented Approach to Non-uniform Cluster Computing, *OOPSLA '05 Proc. 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ACM (2005).

[4] Carlson, W.W., Draper, J.M., Culler, D.E., Yelick, K., Brooks, E. and Warren, K.: Introduction to UPC and Language Specification, *IDA Center for Computing Schences* (1999).

[5] Hara, K., Taura, K. and Chikayama, T.: DMI: A Large Distributed Shared Memory Interface Supporting Dynamically Joining/Leaving Computational Resources, *IPSJ Journal*, Vol.3, No.1, pp.1–40 (2010).

[6] Chen, W.-Y., Iancu, C. and Yelick, K.: Communication Optimization for Fine-grained UPC Applications, *International Conference on Parallel Architecture and Compilation Techniques* (2005).

[7] Su, J. and Yelick, K.: Automatic Support for Irregular Computations in a High-Level Language, *Proc. 19th IEEE International Parallel and Distributed Processing Symposium* (*IPDPS 2005*), p.53 (2005).

[8] Su, J. and Yelick, K.: Array Prefetching for Irregular Array Access in Titanium, *Sixth Annual Workshop on Java for Parallel and Distributed Processing Symposium*, p.158 (2004).

[9] Loveman, D.: High Performance Fortran, *Parallel & Distributed Technology: Systems & Applications, IEEE*, Vol.1, pp.25–42 (1993).

[10] Benkner, S.: Optimizing Irregular HPF Applications using Halos, *Proc. 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, Vol.1586, pp.1015–1024 (1999).

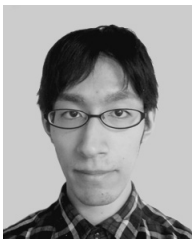
[11] GCC, the GNU Compiler Collection, available from ⟨http://gcc.gnu.org/⟩.

[12] InTrigger, available from ⟨http://www.intrigger.jp/wiki/index.php/InTrigger⟩.

[13] UF Sparse Matrix Collection, available from ⟨http://www.cise.ufl.edu/research/sparse/matrices/⟩.

[14] Nakashima, K.: Lecture Handouts of Computer Science Special Lec-

ture I “Scientific Computational Programming (FEM),” available from ⟨http://nkl.cc.u-tokyo.ac.jp/09s/1D/1D-FEM-1.pdf⟩.



Katsuaki Ikegami is a master’s student at the Department of Information and Communication Engineering, the University of Tokyo. He was born in 1989 and received his B.S. degree from the University of Tokyo in 2011.

Kenjiro Taura is an associate professor at the Department of Information and Communication Engineering, the University of Tokyo. He was born in 1969 and received his B.S., M.S., and D.Sc. degrees from the University of Tokyo in 1992, 1994, and 1997, respectively. His major research interests include parallel/distributed computing and programming languages. He is a member of ACM and IEEE.