Electronic Preprint for Journal of Information Processing Vol.24 No.2

Regular Paper

Introducing a Multithread and Multistage Mechanism for the Global Load Balancing Library of X10

Kento Yamashita[1] Tomio Kamada[1,2]

Received: July 3, 2015, Accepted: October 16, 2015

Abstract: Load balancing is a major concern in massively parallel computing. X10 is a partitioned global address space language for scale-out computing and provides a global load balancing (GLB) library that shows high scalability over ten thousand CPU cores. This study proposes a multistage mechanism for GLB to assign execution stages to tasks and introduces a multithread design into GLB to allow efficient data sharing between CPU cores. The system gives high priority to tasks that are assigned to earlier stages and then proceeds with subsequent stage tasks. When a computing node runs out of tasks at the earliest stage, it requests tasks at the earliest stage from other nodes and awaits responses by processing subsequent stage tasks. When the system identifies the task termination at a certain stage, it executes a reduction operation over nodes. Programmers can define their reduction operations to gather or exchange results of completed tasks. This study provides the implementation method of the extended library and evaluates its runtime overhead using the K computer to a maximum of 256 nodes.

Keywords: dynamic load balancing, X10, GLB

1. Introduction

The field of parallel computing has seen an increase in the number of computing nodes and its application has spread to include computations having irregular features. Load balancing enormous tasks as well as adequate task and communication scheduling to achieve latency hiding have become pressing needs.

Considerable research of dynamic load balancing has been conducted on shared memory computers, and the technology of work-stealing is widely used for task-parallel programs [1], [2]. Research on distributed memory environments is also being conducted. X10 [3], which is a parallel programming language that adopts partitioned global address space (PGAS) model, offers a global load balancing (GLB) library and reveals high scalability over one thousand computing nodes [4], [5]. GLB features a lifeline-based scalable work-stealing algorithm. Results of completed tasks are gathered by means of reduction operations.

However, when the problem requires stepwise progress of computation and intermediate results or their reduced values must be transferred to subsequent tasks, GLB offers few supports for such data sharing among tasks. In addition, GLB does not support multiple workers on a computation node. Moreover, each core of a multicore CPU must be treated as a node. Workers cannot share inputs or computation results until the system completes all tasks.

This study proposes an extension to the GLB library to introduce a multithread design to enable data sharing among workers on a multicore CPU, as well as a multistage mechanism to assign execution stages to tasks. The programmer can create tasks by specifying its execution stages, and the system gives high priority to tasks that are assigned to earlier stages. When a computing node runs out of tasks at the earliest stage, it requests tasks at the earliest stage from other nodes and awaits responses by processing subsequent stage tasks. Each time the system completes all tasks in a stage, it executes a reduction operation over nodes to gather results from the completed stage. Programmers can define their reduction operation according to the target problem and create new tasks based on the results.

The main contributions of this study are a system design and implementation method for introducing multithread and multistage features into a GLB mechanism. Our library adopts an API design in which workers can access their task queues without mutual exclusions. In our load balance algorithm, we use the lifeline-based algorithm of the original GLB for internode load balancing. In addition, we prepare an intranode load balance mechanism that operates with the internode algorithm. For detecting termination of stages, we extend the original detection method of task termination.

Section 2 reviews the system design of the original GLB. Section 3 explains the motivation of this research and Section 4 proposes an API design for multithread and multistage features. Section 5 describes the implementation method of our extended library and Section 6 evaluates its preliminary performance on the K computer. Section 7 reviews related work and Section 8 offers a conclusion to our study.

2. X10 and GLB

X10 is a parallel programming language developed by IBM and adopts PGAS model. The address space is partitioned using **places**, in which each object belongs to any place. `at` (*place*) *S* for statement *S* creates an activity at the *place* and executes *S* in the environment where the values to which *S* refers are copied.

[1] Kobe University, Kobe, Hyogo 657–8501, Japan
[2] RIKEN Advanced Institute for Computational Science, Kobe, Hyogo 650–0047, Japan

X10 features asynchronous programs, and also allows fork-join-style programs. `async` *S* begins a new activity that executes *S*. `finish` *S* awaits the termination of all activities spawned by *S* including those indirectly created.

GLB [5] is a framework of X10. It features a lifeline graph work-stealing algorithm [4]. As the following describes, the API design of GLB allows users to perform fine-grained tasks using low runtime overhead.

The user provides two classes that implement interface `TaskQueue` and `TaskBag`, respectively. `TaskQueue` represents the sequential computation for the problem to be solved and offers a split/merge facility of tasks. `TaskBag` represents a task form used for interplace task transmission. When invoking GLB execution, the user specifies the `TaskQueue` class and a function closure to create a `TaskQueue` instance in each place. `TaskQueue` employs the following methods:

- `process(`*n*`:Long, ...):Boolean` executes task items in the queue until it processes *n* tasks or become empty. The method returns `true` when *n* tasks are completed and returns false otherwise.
- `split():TaskBag` splits tasks in the queue and returns one half as `TaskBag`.
- `merge(TaskBag)` merges the tasks in the received `TaskBag` with tasks in the queue.
- `getResult()` returns the local result in the queue, and `reduce()` reduces the results.

`TaskBag` requires only the `size()` method that returns its size. However, `ArrayListTaskBag`, which is an implementing class of `TaskBag`, prepares the `merge` and `split` methods required for `TaskQueue`. Users can easily define a `TaskQueue` class by extending `ArrayListTaskBag` and adding its `process` method.

Example programs of GLB are provided in the X10 source distribution. In most programs, `TaskQueue` is implemented as a subclass of the sequential class that solves the target problem. The queue exploits the data structure for the sequential program. In the case of unbalanced tree search (UTS), the program processes a depth-first search using a stack, and its task creation/execution corresponds to the push/pop operation of the stack. In UTS, `process(n, ...)` simply executes the search method of the sequential class `n` times, whereas `split()` splits its stack in half. The methods of GLB do not address the creation or execution of each single task, but perform executions or transmissions of tasks in bulk.

The study in Ref. [5] describes the manner in which GLB is applied to two benchmark problems: UTS and betweenness centrality (BC). It then measures performance under weak scalability on Blue Gene/Q and the K computer. With Blue Gene/Q, both problems show nearly a linear increase in speed to a maximum of 16,384 places. On the K computer, UTS scales to a maximum of 4,096 places, whereas BC scales to a maximum of 8,192 places. The current GLB assumes sequential execution of tasks in each place. Therefore, each CPU core is assigned a place during the evaluation.

Regarding the data structure, both programs use `TaskQueue` classes that extend their respective sequential programs and define new `TaskBag` classes. The `TaskQueue` classes store results of completed tasks in array form and gather the results using a reduction operation invoked when GLB terminates. These programs do not share intermediate results during the computation.

3. Motivation and Approaches

This study proposes a multistage mechanism to enable task scheduling in dynamic load balance mechanisms. Some search and simulation problems require stepwise progress of computation in which the loads of subtasks cannot be estimated until they are executed. One assumed application of our library is a program that searches parameter space in a stepwise manner. In order to obtain the optimal result or distribution curve of results, the program first attempts executions on some sampling points. It then recursively examines regions that seem important based on prior executions. Another example is an agent simulation program that assigns high priority to agents that globally affect other agents. It concurrently performs data transfer from these agents and agent processing that shows high locality. A final example involves a situation in which a user wants to process a series of GLB problems in a sequential order, but some of the problems have limited parallelism necessary to utilize computing nodes to the fullest. In this case, for a worker to process tasks of subsequent problems is reasonable when the worker becomes idle. However, sequential execution of GLB problems does not allow concurrent execution of tasks in different GLB problems.

The goal of this study is to design a system that introduces multithread and multistage features into a GLB mechanism and provide an implementation method for the system. Pseudo codes for the aforementioned application examples are shown in Section 4.4. In this study, we have yet to develop actual applications that utilize multithreading or multistage features. In order to complete data sharing among tasks, we must provide a work-stealing scheme that considers data sharing among tasks (discussed in Section 4.4). After providing this scheme, we develop the actual applications and evaluate their performance.

Figure 1 illustrates the manner in which staged tasks are executed in our system. As in the original GLB, a task can dynamically create new tasks. The execution stage of the created tasks must be concurrent with or later than that of the creator task. The current stage of the system is the earliest stage of the uncompleted tasks. Workers give high priority to the tasks that are assigned at earlier stages. When workers in a place complete the that place's earliest stage tasks, the place requests tasks from other nodes. The place awaits responses by processing subsequent stage tasks.

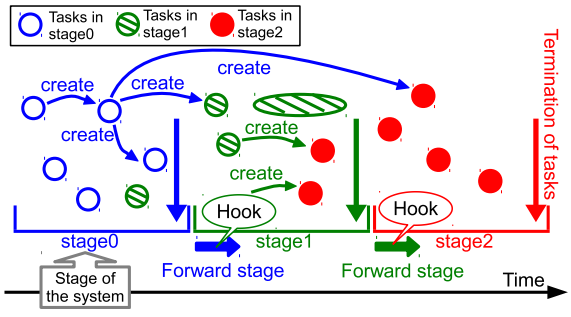


Fig. 1 Stage and tasks.

When the system completes all tasks in a stage, it executes a hook method to gather or exchange results of that stage's tasks. Users can execute reduction operations in the hook and create new tasks based on the results. The system does not confirm termination of the current stage until it completes the hook execution of the previous stage.

In this study, we introduce multithreading with the multistage mechanism to promote data sharing among tasks. Without multithreading, even when all the tasks refer to the same data structure, the system must allocate the same numbers of the copies as the CPU cores on each computing node. In addition, if multithreading allows data sharing among workers on a computing node, a good result identified by a worker can be immediately used by other workers (e.g., for branch and bound of search problems). However, to complete data sharing among tasks, we must prepare work-stealing schemes that consider data sharing among tasks. Therefore, this study designs and implements only a load balancing mechanism that supports multithreading. It remains as a future study to examine work-stealing schemes that consider data sharing.

4. Design for Multithreading and Multistaging

This section describes the proposed API design for multithread and multistage features, and provides some example programs. **Figure 2** illustrates the system design of the extended library.

4.1 TaskQueue and TaskBag

As described in Section 2, GLB does not offer methods for each single task creation or execution. `TaskQueue` represents a set of tasks and provides the process, split, and merge methods. In many programs, the `TaskQueue` class is implemented as a subclass of the sequential version of the target problem.

In the extended library, each task is assigned to its execution stage, but our library does not offer methods to assign each task to its stage. Instead, the staged versions of `TaskQueue` and `TaskBag` manage the stage information of their tasks.

Regarding multithreading, `TaskQueue` continues to represent sequential computation. The system allocates the specified number of worker threads in each place and prepares the same number of worker queues. The default number of worker threads is `X10_NTHREADS`. The system also prepares two shared queues in each place. These queues are used for inter- and intraplace load balancing.

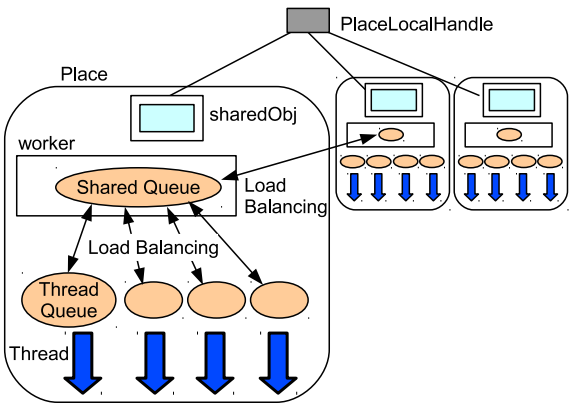


Fig. 2 System design for multithreading.

4.2 Shared Data Structures and Stage Transition

This study also aims to enable data sharing among tasks. In our library, users can place a shared object (called sharedObj) at each place. The `PlaceLocalHandle` class of X10 organizes these objects into a distributed object. As the function closure to build `TaskQueue` receives the reference to sharedObj as a parameter, worker threads can store their data in sharedObj. Data sharing between places is represented by the reduction operation over sharedObjs. Users can define the `reduce` method of the sharedObj. They can use array reduction methods of the `Team` class, which utilizes collective communications facilities. We also plan to provide reducible distributed objects that represent major data types, such as primitive data types or `HashMap`. These distributed objects can be used as sharedObj. The details concerning distributed objects is out of the scope of this study.

Users define a hook method that is invoked at program completion or stage transitions. The system first invokes the hook method of worker queues to gather computation results into sharedObj, and then executes the `reduce` method of the sharedObj. The `terminationHook` method of `TaskQueue` is invoked when the system determines that all tasks have been completed. `TaskQueueStaged` is the staged version of the `TaskQueue` interface. The `forwardHook` method of `TaskQueueStaged` is invoked at stage transitions.

The `terminationHook` methods of both thread and shared queues in each place are executed in a sequential manner. By contrast, the `forwardHook` methods for stage transitions are executed by respective worker threads to prevent concurrent access to the worker queue and maintain data consistency. The `reduce` method of sharedObj is executed in each place after the `terminationHook` or `forwardHook` methods terminate. The stage transmission is completed when the reduction operation terminates.

4.3 Modifications in API

This section summarizes the modification points of API. Because `TaskQueue` is also used for shared queues, it must employ `removeAll():TaskBag` to remove all of its tasks. For compatibility with `TaskQueueStaged`, it also offers `hasJob()`, which returns `true` when it has tasks and return false otherwise. As previously described, the reduction operation is managed by the `terminationHook` method of `TaskQueue` and `reduce` method of sharedObj, instead of by the `getResult` and `reduce` methods of `TaskQueue` in the original GLB.

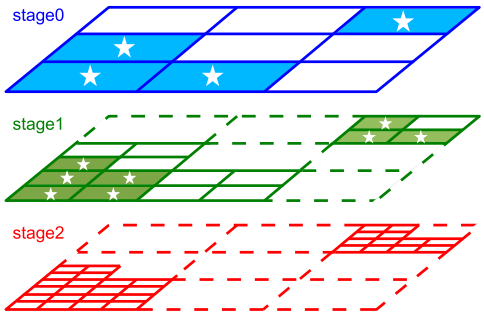
`TaskQueueStaged` extends `TaskQueue` and includes `jobStage()`, which returns the earliest stage value of its tasks. The stage value is an integer incremented from zero. To identify the current stage of the system, the `process` method receives the value as a parameter called `stage`. The `process(n, stage, ...)` method can exit when it runs out of the earliest stage tasks even if the number of executed tasks is fewer than `n`. The return value of `jobStage()` must not decrease by means of the `process` execution. It can decrease only when it receives the earlier tasks by means of `merge`. The `split` and `removeAll`

methods of `TaskQueueStaged` split or remove, respectively, only the earliest stage tasks in the queue and their return value types both become `TaskBagStaged`, which contains stage information about the contained tasks. This type is also used as an input parameter of the `merge` method.

4.4 Program Examples

This section provides program examples (pseudo codes) that realize the applications described in Section 3 and describes the manner in which programmers use our library.

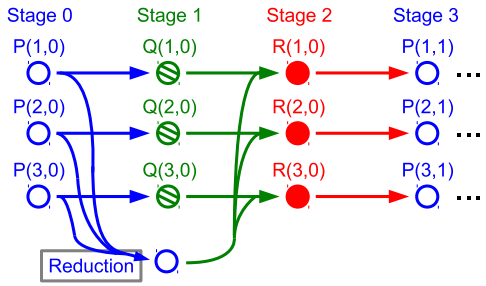
The first example is a program that searches for the optimal result in a parameter space using a stepwise approach. The upper part of **Fig. 3** illustrates the stepwise search, and the lower part shows the pseudo code of the program. The program first divides the space into several regions and performs a sampling evaluation of the regions. It then recursively divides the regions that seem important based on prior executions. `Queue` (line 1) implements `TaskQueueStaged` and includes its tasks in a list called `stagedTasks` (line 2), in which each element represents a set of tasks in the corresponding stage. The cutoff threshold is stored in a sharedObj called `sharedLimit` (line 3). When a worker finds a good result, other worker threads in the place can use the threshold value, and workers in other places can use the value from the next stage. In the GLB setup, function closures to create `Queue` and sharedObj are specified (lines 28–31). The variable `jobStage` at line 4 is used for `jobStage()`. In the `process` method, the worker executes a maximum of `n` tasks in the earliest stage. If the result of a task passes the threshold, the task prepares its sub-regions and creates corresponding new tasks in the next stage. When `Queue` runs out of tasks for the earliest stage, it increments the value of `jobStage` (line 19). In the `merge` method, `Queue` decrements `jobStage` according to the stage of the received tasks. The threshold is confirmed during task executions (lines 11–12) and task creations (lines 13–16). In the stage transition, the system calls the `reduce()` method of `sharedLimit:DistValueMin`, and the threshold value is set to the minimum value among places.




```
1  class Queue implements TaskQueueStaged[Queue] {
2    val stagedTasks: List[List[Task]];
3    val sharedLimit: DistValueMin[Double];
4    var jobStage:Long;
5    /* omitting constructor, split, merge, and jobStage */
6    public def process(n:Long, stage:Long, ..):Boolean {
7      val stage0 = stagedTasks.get(jobStage);
8      val stage1 = stagedTasks.get(jobStage+1);
9      while(n>0 && !stage0.isEmpty()) {
10       val cand = stage0.removeLast();
11       if(cand.score < sharedLimit.getValue()) {
12         val result = tryCalc(cand);
13         if(result.score < sharedLimit.getValue()) {
14           atomic { updateLimit(result.score);}
15           stage1.addAll(result.getSubRegions());
16         }
17       }
18     }
19     if(stage0.isEmpty()) jobStage++;
20     return true;
21   }
22   public def forwardHook(shared:TeamReducible,
23               stage:Long, terminated:Boolean){
24     /* do nothing */
25   }
26 }
27 /* main routine */
28 GLB.setupStaged[Queue](
29   ()=>{/*sharedLimit*/ return new DistValueMin[..](); },
30   (sharedLimit:TeamReducible)=>
31       {/*Queue*/ return new Queue(sharedLimit); }, ...);
```

Fig. 3 Use case (stepwise search).

The other example is a simulation program in which each step of the program includes three computation parts: P, Q, and R. The upper part of **Fig. 4** illustrates the dependency among the parts.




```
1  class Queue implements TaskQueueStaged[Queue] {
2    val shared:MyShared;
3    var pSum:Long;
4    public def process(n:Long, stage:Long, ..):Boolean{
5      if(!hasJob()) return false;
6      if(jobStage()-stage>1) return false;
7      switch(jobStage()%3) {
8        case 0: return processP(n);
9        case 1: return processQ(n);
10       case 2: return processR(n,stage);
11     }
12   }
13   public def processP(n:Long): Boolean {/* same as Q */
14     while(n>0 && !taskP.isEmpty()) {
15       val a = taskP.removeLast();
16       taskQ.push(calcP(a));
17     }
18     if(taskP.isEmpty()) jobStage++;
19     return true;
20   }
21   public def processR(n:Long, stage:Long): Boolean {
22     if(stage-shared.reducedStage.get()<2) return false;
23     /* task processing */
24   }
25   public def forwardHook(shared:Any, stage:Long, ..){
26     if(stage%3==1) atomic { /* P->Q */
27       shared.pSum.add(this.pSum); shared.stage=stage;
28     }
29     /* setup of the new stage */
30   }
31 }
32 class MyShared implements TeamReducible {
33   var pSum: DistValueSum[Double];
34   var stage:Long;
35   var reducedStage:AtomicLong = new AtomicLong(-1);
36   void reduce(team:Team)  {
37     if(stage%3!=1) return;
38     pSum.reduce(team);
39     reducedStage.set(stage);
40   }
41 }
```

Fig. 4 Use case (dependency between stages).

We denote the computation parts of step i for element $a \in U$ as task P(a, i), Q(a, i), and R(a, i). In this program, Q(a, i) requires the results of P(a, i), and R(a, i) requires the result of Q(a, i) and the reduction value of $\forall b \in U$ P(b, i). In the next stage, P(a, $i + 1$) requires the result of R(a, i). In our library, the user can assign task P(a, i), Q(a, i), and R(a, i) to stage $3 \times i$, $3 \times i + 1$, and $3 \times i + 2$, respectively. P(a, i), Q(a, i), and R(a, i) create Q(a, i), R(a, i), and P(a, $i + 1$), respectively, in the next stages. R(a, i) must be blocked until the system reduces the results of $\forall b \in U$ P(b, i). The lower part of Fig. 4 shows pseudo code used to realize the aforementioned behavior.

`Queue` executes P, Q, and R according to the `jobStage` (lines 7–11) and creates the subsequent tasks (e.g., line 16). `Queue` prepares `pSum` (line 3) to store the reduced results of the P tasks executed by the worker thread. `MyShared` is defined for sharedObj (lines 32–41). In the stage transition from P to Q, the value of `pSum` is included in `shared.pSum` (lines 26–28) and reduced among places (lines 36–40). Executing R requires confirming that the reduction on P (line 22) has completed. In addition, workers only execute tasks in the current or next stage (line 6). If tasks in the stage after the next are executed, tasks of P in different steps may be mixed in the queue called `taskP`.

Finally, we consider concerns resulting from task transfer. The aforementioned programs assume that each task contains all data required for program execution. If P(a, i) is transferred to another place, the required data are also copied to the destination place, and Q(a, i) is created in the place. Suppose the program creates P(a, i) and Q(a, i) simultaneously and they share local data for element a. If one of these two tasks is transferred to another place, the tasks cannot continue the data sharing. To continue the data sharing, the system must move both tasks and their shared data. If the task needs to access neighboring elements of a, we must consider the locality of tasks when splitting the tasks in a queue. In a future study, we plan to use distributed objects to manage data distribution over places. We also intend to enable coordination between dynamic load balancing and data sharing on the distributed object.

5. Implementation

This section describes the implementation methods we employed to introduce multithread and multistage mechanisms into GLB. The major concerns of implementation are efficient dynamic load balancing and detecting program and stage termination.

5.1 Load Balancing

As the original GLB shows high scalability on dynamic load balancing among places, the implementation of our library adopts the same interplace load balancing policy. In addition, we use shared queues for intraplace load balancing.

In the work-stealing algorithm of the original GLB [4], when the worker of a place runs out of tasks, the algorithm first tries to steal tasks from a randomly selected place several times. It then starts work-stealing through the lifeline graph. When a place receives a load request and has sufficient tasks, the place calls `split()` in its queue and returns a half of its tasks. When the place does not have sufficient tasks, the place immediately returns the failure of the request. In the case of a lifeline request, the requested place remembers the request and gives half of the tasks when the place subsequently creates or receives enough tasks.

When introducing multithreading, we retain the policy that returns half of the tasks in a place when that place receives a load request. To implement this policy, we use two shared queues, sharedQ and sharedQ2. SharedQ is designed to hold nearly half of the tasks in a place and is used for interplace load balancing. SharedQ2 is mainly used for intraplace load balancing. The following algorithms are used to hold the task volume of sharedQ. The first is the algorithm that is not concerned with multistaging. The second is the multistage version. In both algorithms, we only use mutual exclusion in worker access to shared queues. By contrast, each worker queue is accessed only by its owner thread without mutual exclusions.

- When receiving an interplace load request, the place basically sends all tasks in its sharedQ. However, before sending the tasks, the place examines sharedQ2 and moves half of the tasks to sharedQ2 when sharedQ2 is empty.
- When the sharedQ at a place becomes empty, all worker threads in the place are notified of this situation.
- Each worker thread performs the following load checks before executing `process()`. First, the worker thread determines whether the aforementioned notifications have been received and then sends half of its tasks to sharedQ when the notifications are received. The worker next examines sharedQ2 and gives half of its tasks to sharedQ2 if sharedQ2 is empty. However, when the worker has no tasks, the worker steals half of the tasks in sharedQ2. When `split()` returns `null`, the worker steals all the tasks of sharedQ2. When `removeAll()` also returns `null`, the worker splits the tasks of sharedQ, moves half to sharedQ2, and reattempts work-stealing from sharedQ2.

We next describe the algorithm for staged tasks. A place or thread steals tasks of a victim only if the value of `jobStage()` is greater than that of the victim. Half of the tasks from the earliest stage in the victim queue can be stolen using `split()`. In the case of interplace load balancing, our algorithms also move tasks in the next earliest stage of the victim in order to distribute them.

- When receiving an interplace load request, the place compares stage $s1$ of its sharedQ with stage $s0$ of the sharedQ at the requesting place, and sends tasks only when $s0 > s1$. When sending tasks, the place first calls `removeAll()` to remove all the tasks in the earliest stage of its sharedQ, then calls `split()` to remove half of the tasks from the next earliest stage if $s0 > s1 + 1$ and sends them to requesting places. To guarantee volume of tasks exists in the requested place, the place examines stage $s2$ of the sharedQ2 prior to the aforementioned procedures. When $s1 < s2$, the place splits the tasks in the earliest stage of sharedQ and merges half to sharedQ2.
- When a place sends the task in its sharedQ to another place, all worker threads in the place are notified of this situation.
- Each worker thread conducts the following load checks before executing `process()`. First, the worker thread con-

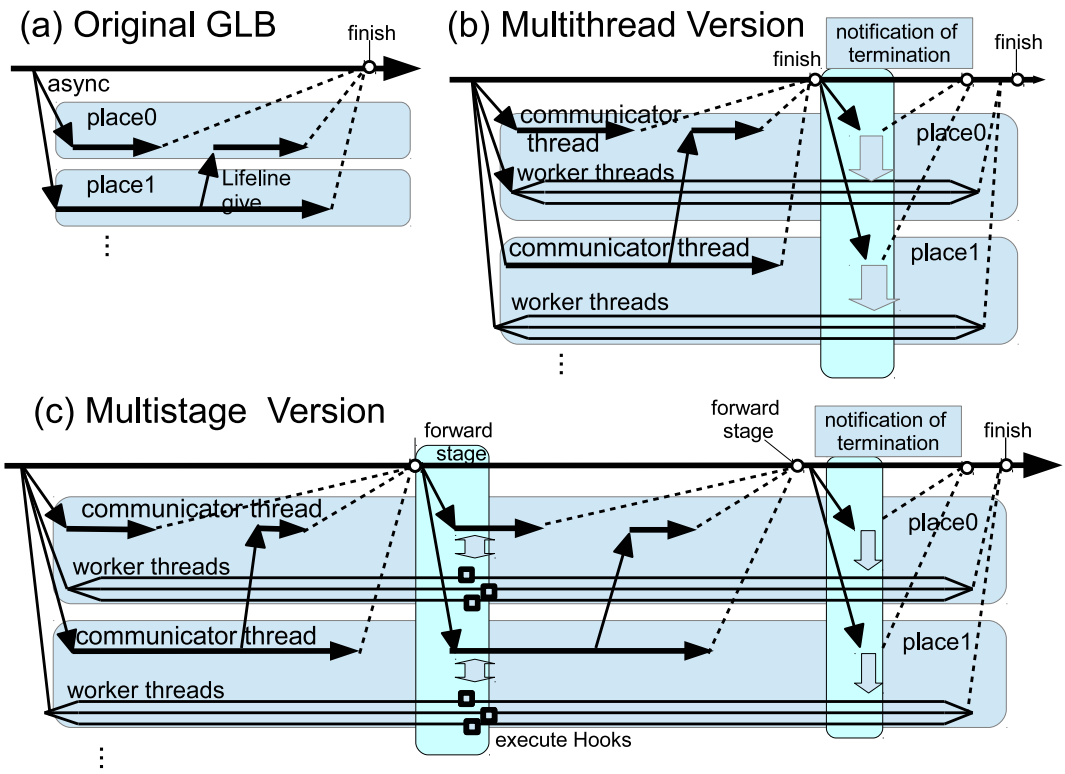


Fig. 5 Termination detection of tasks and stage migration.

firms whether the aforementioned notifications have been received. When received, the worker calls `split()` to split its tasks and merges the tasks into sharedQ. The worker then compares the value of `jobStage()` with that in sharedQ2. When the stage value of sharedQ2 is greater than that of the worker queue, the worker splits its tasks and assigns half to sharedQ2. When the stage value of sharedQ2 is smaller than that of the worker queue, the worker splits the tasks of sharedQ2 and steals half.

5.2 Termination Detection

The original GLB detects whether tasks have terminated using `finish` operations [4]. As described in Section 5.1, the place that receives a task request immediately returns an answer regarding whether it has sufficient tasks to give (**Fig. 5** (a)). When a requesting place fails to receive tasks from randomly selected places and lifeline buddies, the worker in the place terminates its activity. By contrast, the place that receives a lifeline task request remembers the request and then sends tasks when it has a sufficient amount. When the receiving place for tasks possesses no worker activities, the place creates a new activity as a worker thread using `async`. The original GLB detects the termination of such worker activities using `finish`. It determines whether all GLB tasks have been terminated, and then starts reducing computation results.

To introduce multithreading, we prepare two types of threads in each place: (1) a communicator thread used for message probing and termination detection, and (2) other worker threads that concentrate on task processing. Figure 5 (b) illustrates the system behavior of the termination detection algorithm that is not concerned with multistaging (simply called the multithread version). Worker threads are spawned in each place at the beginning of the process. These threads are notified of the termination of GLB (or notified of stage transitions in the multistage version) by communicator threads via shared memory variables. The activity of a communicator thread is created when a place receives a task response to a lifeline request and has no active communicator thread. The activity is terminated when the place runs out of tasks. To inform the communicator whether worker threads in the place have tasks, worker threads prepare shared variables to record task information (stage information in the multistage version) at every load check.

In the multistage version (see Fig. 5 (c)), the aforementioned detection method is used to identify termination of tasks in each stage. The system completes its activity when it attempts to start a new stage and confirms that no task execution or receipt exists since when each place ran out of tasks in the previous stage and all load requests and responses to lifeline requests have been acknowledged. When the system starts the termination process, each communicator first notifies workers of this process, awaits termination by workers, invokes `forwardHook` methods for all queues, and begins a reduction operation on sharedObj. In the case of stage transitions, each communicator notifies workers of the stage transition to allow them to execute `forwardHook` of their queue. After confirming executions, the communicator executes `forwardHook` of both its communicator and shared queues, and then starts a reduction operation on sharedObj. While the tasks of the next stage are executed during the stage transition, the stage transition process of the next stage is blocked until the transition from the previous stage has been completed.

5.3 Utility for Multistaging

We provide a utility class that uses stages to execute

a series of GLB programs in a sequential order. The class implements `TaskQueueStaged` and contains an array of `TaskQueue` objects. Each `TaskQueue` element corresponds to a stage. The `TaskQueueStaged#forwardHook` method invokes `terminationHook` methods for the corresponding `TaskQueue` elements. The programmer can gather the results of each stage and use them in the following stages.

6. Preliminary Evaluation

This section reports experimental performance results of our extended library on the K computer. The K computer is a supercomputer at the RIKEN Advanced Institute for Computational Science. Each computing node has one scalar CPU (SPARC64™ VIIIfx, 8 cores) and 16 GB of memory.

When using the multithread and multistage versions of our library, we allocated one place per node using `X10_NTHREADS=8`. The program is compiled by means of native X10 version 2.5.1 and executed with the MPI version runtime (`-x10rt=mpi`). The backend compiler is Fujitsu C/C++ Compiler version K-1.2.0-18 with `-Xg` and `-Kfast` options. This study also uses the performance values of the original GLB measured in Ref. [5]. For the measurement in Ref. [5], we allocated eight places per node (one place per core). We used native X10 version 2.4.0 and Fujitsu C/C++ Compiler version 1.2.0-14, using nearly the same compiler options as in this measurement.

In this experiment, we used the UTS program that is provided as a sample program of GLB. The multithread version has two modifications from the original. The first is in the manner of gathering computation results. The multithread version uses sharedObj to gather the results as described in Section 4. The second is padding for the `TaskQueue` class. Without padding, the `process` method of the multithread version is 20% slower than the original. This implies that we must prepare cache-conscious object representation to gain the benefit of shared memory.

The staged version uses the utility library described in Section 5.3, and executes four stages, each of which corresponds to a program execution of UTS. As explained in detail in a later paragraph, the original GLB and multithread version of the UTS show high efficiency. Therefore, the staged version shows a slight possibility that it can be faster. This comparison allows us to measure the overhead of the multistage mechanism.

In our library, the communicator thread is also designed to process tasks that call `process` methods. However, our current implementation encountered some network response problems when the communicator threads executed `process()`. To avoid these problems, we added an option for communicator threads to skip `process()` calls. Therefore, the multithread and staged versions can use only seven threads to execute `process` methods in these experiments.

Figure 6 shows the performance values to a maximum of 256 nodes. We measured the performance in weak-scaling using the same execution parameter as in Ref. [5]. The original and multithread versions completed the execution in 5–10 minutes, and the staged finished in 20–40 minutes. The Y-axis in Fig. 6 represents the number of traversed nodes in UTS per second. All the versions reveal high scalability to a maximum of 256 nodes.

	execution parameter								
# of nodes	1	2	4	8	16	32	64	128	256
d	16	16	17	17	18	18	19	19	20

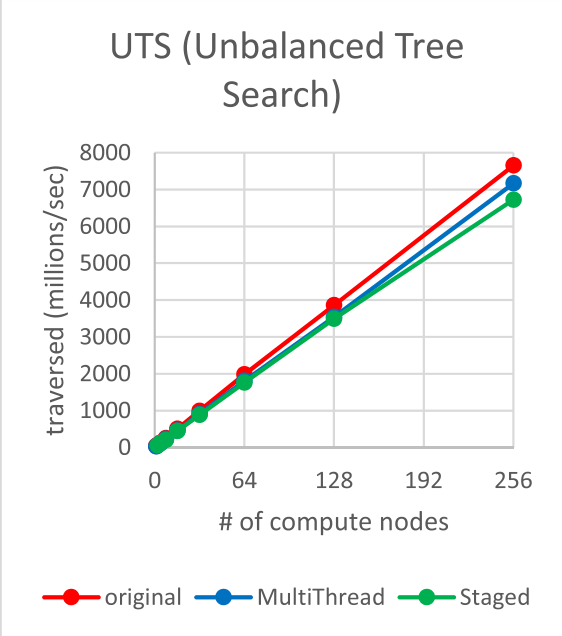


Fig. 6 Performance evaluation.

The performance values become somewhat worse when applying both multithreading and multistaging.

The multithread version is slower than the original by 7–10%. This is because the number of working threads is changed from eight to seven, and we are currently attempting to identify the cause of this network response slowdown.

The staged version is slower than the multithread version by 3% or less in most cases, and by 12% and 7% in the case of eight and 256 nodes, respectively. Although performance by the staged version has a slight chance to improve in this experiment, the results show that the current staged version is slower than in multiple invocations of the non-staged version. We plan to reduce the runtime overhead for staging facilities.

In this experiment, we only use 256 nodes of the K computer because some network performance problems have been identified when using broadcasts over 256 nodes and their `finish` operations. The problem does not occur when our library is not used and we are therefore attempting to locate the cause of the problems.

This experiment only evaluates overhead when introducing multithreading and multistaging features. This is because we have yet to develop actual applications that utilize multithreading or multistaging. As described in Section 4.4, we are currently working to enable coordination between dynamic load balancing and data sharing over multiple nodes. We plan to develop actual applications and evaluate its performance in a future study.

7. Related Work

Much research has been conducted on dynamic load balancing especially using task-parallel-style parallel programming lan-

```
1 finish { /* using signal-wait-next */
2   phaser ph = new phaser();
3   for(j=1; j<=n; j++) {
4     async phased {
5       for(..) {
6         /* calcP */
7         signal;
8         /* calcQ */
9         next { reduction() }
10        /* calcR */
11      }
12    }
13  }
14 }
15 finish { /* using signal/wait-only */
16   phaser[] phs = new phaser[n+2];
17   for(j=1; j<=n; j++) {
18     async phased(signalOnly(phs[j]),
19            waitOnly(phs[j-1], phs[j+1])) {
20       for(..) {
21         /* calcP */
22         signal; // signal phs[j]
23         /* calcQ */
24         next; // Await signals from iter j-1 & j+1
25         /* calcR */
26       }
27     }
28   }
29 }
```

Fig. 7 Code example of Phaser.

guages [1], [2]. These languages support natural expressions of fork-join parallelisms and enable their synchronization. The language Cilk offers constructs for `spawn` and `sync`, whereas Chapel offers a series of `begin` statements for task creations and sync type variables for synchronization.

Barrier synchronization is commonly used to synchronize parallel threads (or processes) of data-parallel programs. In the case of an ordinal barrier, any thread that reaches the barrier point must wait until all threads reach that point. To reduce the waiting time for synchronization, Fuzzy Barrier [6] introduces barrier regions. A thread can exit a region when all threads reach the first instruction of the region.

X10 provides *clocks* that can be used for barrier synchronization. The programmer can create an activity that is registered with a clock `c` by the statement `async clocked(c)` S, and dynamically unregister the activity from the clock by `c.drop()`. The registered activity of `c` executes `c.advance()` to process barrier synchronization with other registered activities of `c`, and the activities can enter the barrier region by `c.resume()`.

Phaser [7] is a new synchronization construct that unifies collective and point-to-point synchronization. Phaser is proposed as an extension for X10 and used for synchronization among activities in the same place. Activity is registered with a phaser in one of four modes. Lines 1–14 of **Fig. 7** show a sample case using `signal-wait(-next)` modes to enable a fuzzy barrier synchronization, which corresponds to the second example in Section 4.4. When using `signal/wait-only` modes, we can describe point-to-point synchronizations as in lines 15–29. This example uses an array of phasers, `phs`, and the j-th activity signals `phs[i]` and then waits for the signals on `phs[i-1]` and `phs[i+1]`. Using these phaser modes, programmers can naturally describe various kinds of synchronization. Phaser also features safety properties, such as deadlock freedom and phase-ordering. Phaser can be used in Habanero-Java [8].

Comparing Phaser with our multistage mechanism, Phaser allows barrier or producer-consumer synchronizations among activities, and the programmer can set data dependency between operations. However, Phaser cannot specify priority among concurrent tasks. If a programmer wants to give higher priority to the computation of P rather than Q, he or she must set explicit dependency between them. By contrast, our library does not offer syntax extension of X10, and the programmer must divide the problem into separately described tasks. Using our multistage mechanism, the programmer can easily specify global priority of tasks. However, the dependency between tasks must be implemented through task creation relationship or guard conditions for task executions, as in the second program described in Section 4.4. Phaser has an obvious advantage in its natural description of synchronizations.

However, Phaser is designed for shared memory environments and evaluated on large-scale symmetric multiprocessor systems, but it does not support distributed memory environments. When the programmer wants to use a phaser array to describe dependency of tasks, he or she must consider the distribution of phaser elements over nodes, while considering dynamic task relocation. In our library, the programmer must describe data dependency among tasks explicitly, but he or she can easily define task priority using the stage mechanism. The system realizes global scheduling of tasks, offering load balancing and termination detection of staged tasks.

8. Conclusion

This study proposed an extension to the GLB library, in order to introduce multithread and multistage features. We presented our system design and implementation method. The system reveals high scalability to a maximum of 256 nodes. However, implementation problems remain that appear to be caused by network message scheduling, and the system currently requires one thread that concentrates on message processing at each computing node. We plan to release our library after correcting these problems. In a future study, we plan to enable coordination between dynamic load balancing and data sharing over places and develop actual applications that utilize multithread or multistage features.

Acknowledgments This work was supported by CREST, JST. Part of the results is obtained by using the K computer at the RIKEN Advanced Institute for Computational Science.

References

[1] Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H. and Zhou, Y.: Cilk: An Efficient Multithreaded Runtime System, *SIGPLAN Not.*, Vol.30, No.8, pp.207–216 (online), DOI: 10.1145/209937.209958 (1995).

[2] Cray Inc.: *Chapel Language Specification Version 0.91* (2012).

[3] Saraswat, V., Bloom, B., Peshansky, I., Tardieu, O. and Grove, D.: *X10 Language Specification Version 2.5* (2015).

[4] Saraswat, V.A., Kambadur, P., Kodali, S., Grove, D. and Krishnamoorthy, S.: Lifeline-based Global Load Balancing, *Proc. 16th ACM Symposium on Principles and Practice of Parallel Programming, PPoPP '11*, New York, NY, USA, pp.201–212, ACM (online), DOI: 10.1145/1941553.1941582 (2011).

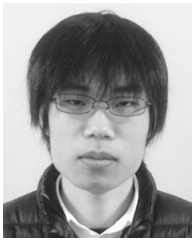
[5] Zhang, W., Tardieu, O., Grove, D., Herta, B., Kamada, T., Saraswat, V. and Takeuchi, M.: GLB: Lifeline-based Global Load Balancing Library in X10, *Proc. 1st Workshop on Parallel Programming for Analytics Ap-*

plications, *PPAA '14*, New York, NY, USA, pp.31–40, ACM (online), DOI: 10.1145/2567634.2567639 (2014).

[6] Gupta, R.: The Fuzzy Barrier: A Mechanism for High Speed Synchronization of Processors, *Proc. 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, *ASPLOS III*, New York, NY, USA, pp.54–63, ACM (online), DOI: 10.1145/70082.68187 (1989).

[7] Shirako, J., Peixotto, D.M., Sarkar, V. and Scherer, W.N.: Phasers: A Unified Deadlock-free Construct for Collective and Point-to-point Synchronization, *Proc. 22nd Annual International Conference on Supercomputing*, *ICS '08*, New York, NY, USA, pp.277–288, ACM (online), DOI: 10.1145/1375527.1375568 (2008).

[8] Cavé, V., Zhao, J., Shirako, J. and Sarkar, V.: Habanero-Java: The New Adventures of Old X10, *Proc. 9th International Conference on Principles and Practice of Programming in Java*, *PPPJ '11*, New York, NY, USA, pp.51–61, ACM (online), DOI: 10.1145/2093157.2093165 (2011).



Kento Yamashita received his B.E. degree from Kobe University in 2015. Now he is a student of the graduate school of Kobe University. His research interest is dynamic load balancing on large-scale distributed-memory computers.

Tomio Kamada received his B.E. and M.E. degrees from The University of Tokyo in 1993 and 1995, and received his Ph.D. degree from Kobe University in 2004. He was a research associate at Kobe University during 1998–2010, and became a lecturer in 2010. His research interest includes parallel and distributed computations, and the runtime systems of programming languages.