Preliminary Evaluations of Probabilistic Guards for a Work-Stealing Framework

Hiroshi Yoritaka*, Ken Matsui‡§, Masahiro Yasugi†, Tasuku Hiraishi‡ and Seiji Umatani‡
*Kyushu Institute of Technology, Fukuoka, Japan
Email: hotsoup_p@ai.kyutech.ac.jp
†Kyushu Institute of Technology, Fukuoka, Japan
‡Kyoto University, Kyoto, Japan
§Presently with Nintendo Co., Ltd.

***Abstract*—We propose probabilistic guards and analyze their performance. In order to reduce the total task division cost, probabilistic guards can prevent thief workers from stealing small tasks from victim workers probabilistically. In this study, we implemented probabilistic guards on a work-stealing framework called Tascell and conducted a preliminary evaluation. We confirmed the good performance of probabilistic guards through the preliminary evaluation. In theory, a thief may repeat an unbounded number of probabilistically prevented steal attempts until success if a victim employs a probabilistic guard that rejects steal attempts with a non-zero probability. Therefore, in this study, we examined a mechanism that invalidated probabilistic guards on demand by setting an upper limit to the number of repeated, probabilistically prevented steal attempts. We evaluated its potential effects on probabilistic guards by measuring actual numbers of repeated attempts until success. We also evaluated performance of probabilistic guards with various upper limits.**

***Index Terms*—parallel programming languages; work stealing; probability; concurrency; many-core; Barnes-Hut algorithm**

I. Introduction

The proliferation of parallel computing environments, including multicore processors, has resulted in the active investigation of parallel programming frameworks. An important objective is to enable users to easily achieve high parallel performances for a wide range of applications. For example, in many irregular applications, it is difficult to statically divide the entire work into parallel tasks, each with an equal amount of work. Dynamic load balancing is generally effective in parallelizing such applications, where a task is dynamically allocated to an idle worker.

Multithreaded languages [1], [2], [3] provide efficient dynamic load balancing based on work-stealing techniques. Cilk [1] is a successful pioneer language among such languages. It provides effective load balancing for many applications, including irregular ones, that is, it keeps all workers busy by creating plenty of "logical threads" and adopting the oldest-first work-stealing strategy. Cilk employs an implementation technique called Lazy Task Creation (LTC) [4], in which each worker spawns plenty of logical threads and schedules them internally, and thus, efficiently. Cilk allows an idle worker to steal a logical thread from another worker. Here, logical threads are used as tasks dynamically allocated to idle workers. When a logical thread recursively spawns offspring logical threads, the oldest-first work-stealing strategy is generally effective in making tasks larger.

Recently, a logical-thread-free parallel programming/execution framework called Tascell was proposed as an efficient work-stealing framework [5]. Tascell implements *backtracking-based load balancing* with on-demand concurrency. A worker performs a computation sequentially, unless it receives a task request with polling. When requested, the worker spawns a real task by temporarily backtracking and restoring its oldest task-spawnable state. Because no logical threads are created as potential tasks, the cost of managing a queue for them can be eliminated. Tascell also promotes the long-term reuse of workspaces, and it improves the locality of reference because it does not have to prepare a workspace for each concurrently runnable logical thread.

In this study, we propose a mechanism that probabilistically guards workers (victim workers) from idle workers (thief workers) [6]; we can skew the probabilities of eventual success in repeated uniformly random steal attempts among victim workers to be non-uniform as the "relative" values of the probabilities of success in a single steal attempt among victim workers. Although the Tascell framework provides efficient load balancing with on-demand concurrency, dividing small tasks causes the degradation of parallel performances in some applications. For example, in order to divide a task, the necessary data for the computation must be copied. Moreover, in order to merge multiple results of dynamically divided tasks, workers may have to execute complicated operations which are hard to parallelize properly and efficiently. In such applications, existing random work stealing may result in performance degradation because of the overheads of dividing and merging tasks. Probabilistic guards can reduce the total task division cost and mitigate such performance degradation by preventing thief workers from stealing small tasks probabilistically. Unlike thresholds, probabilistic guards reduce average concurrency without loss of potential concurrency.

We applied the concept of probabilistic guards to the Barnes-Hut algorithm [7]. The Barnes-Hut $O(N \log N)$ tree algorithm is widely used for N-body simulations. It constructs a tree that represents the hierarchical spatial structure of bodies to calculate forces exerted on each body. There are various algorithms to parallelize the Barnes-Hut algorithm. We observed that dividing small tasks cause substantial performance degradation, and that we can preserve the overall performance by employing probabilistic guards.

In probabilistic guards, when a thief worker fails to steal, it repeatedly tries to steal a task from victim workers. In theory, a thief may repeat an unbounded number of probabilistically prevented steal attempts if a victim employs a probabilistic guard that rejects steal attempts with a non-zero probability. Therefore, we examined a mechanism that invalidated probabilistic guards on demand by setting an upper limit to the number of repeated, probabilistically prevented steal attempts. We evaluated its potential effects on probabilistic guards by measuring actual numbers of repeated attempts. We also evaluated the performance of probabilistic guards with various upper limits.

The contributions of this paper are as follows:

- We propose the concept of probabilistic guards, which prevent thief workers from stealing a task from victim workers probabilistically and mitigate performance degradation by reducing the total task division cost for actual concurrency in the Tascell framework.
- We present parallelized implementations of the Barnes-Hut N-body simulation, in which probabilistic guards are applied.
- We present the preliminary evaluation results of the parallelized Tascell programs. Experimental results on many-core processors indicate good performance improvement in force calculation of the Barnes-Hut algorithm.
- We propose a mechanism that invalidates probabilistic guards on demand by setting an upper limit to the number of repeated, probabilistically prevented steal attempts. We examined the effects of the upper limit on probabilistic guards. We also present the preliminary evaluation results of probabilistic guards with various upper limits.

This paper is organized as follows. First, we provide a brief description of the Tascell framework in Section II. Then, we explain the concept of probabilistic guards, upper limits, and an extension to the Tascell framework in Section III. In Section IV, we show parallel implementations of the Barnes-Hut simulation as an example that uses probabilistic guards. Finally, we present the evaluation results of parallel implementations: with probabilistic guards, with thresholds, and without probabilistic guards and thresholds.

II. Tascell Framework

The Tascell framework [5], [8] consists of a compiler for an extended C language called Tascell language, and a runtime system for parallel computations.

In Tascell, computations are accomplished by the Tascell workers that execute tasks. A task is a data object that is necessary for accomplishing a certain computation. Its structure is defined in a Tascell program by users. A task is associated with a specific function. When a worker receives a task, it invokes the associated function and completes its work with the given task object. Tascell employs a randomized work-stealing strategy to achieve dynamic load balancing among the workers. In Tascell, an idle worker (thief) can request a task from a loaded worker (victim). When receiving a task request, the victim worker creates a new task by dividing its own task and returns it to the thief. Then, the thief performs the received task and returns its result to the victim worker.

A Tascell worker spawns a task by temporarily backtracking and restoring its oldest task-spawnable state. That is, when a worker receives a task request,

1) it temporarily backtracks (goes back to the past),
2) it spawns a task (and changes the execution path to receive the result of the task),
3) it returns from the backtracking, and
4) it resumes its own task.

The Tascell worker always chooses not to spawn a task *at first* and performs sequential computations. However, when the worker receives a task request, it spawns a task as if *it changed the past choice.*

In general, we can spawn a larger task by backtracking to the oldest task-spawnable state. Because no logical threads are created as potential stealable tasks, we can eliminate the cost of managing a queue for them in Tascell. Moreover, we can parallelize some highly serial applications in a straightforward manner, in which a worker continuously and serially updates a single workspace for computations, because Tascell has the following characteristics:

- While a Tascell worker performs a sequential computation, it can reuse a single workspace, whereas a logical thread typically requires its own workspace.
- When a new task is spawned, the victim's workspace can be copied for the thief. Because a task is spawned only when it is requested by idle workers, workspace copying can occur only when it is actually needed.

III. Our Proposal

The Tascell framework may need to copy the necessary data for the thief worker's computation from the victim's workspace. The copying cost is included in the task division cost. If the copying cost becomes large compared with the amount of the stolen work, dividing *small* tasks causes performance degradation in parallel execution since it increases the total task division cost. The task division costs may also include costs to merge results of divided tasks. While many multithreaded languages require these division operations *eagerly* when a concurrently runnable logical thread is spawned, Tascell requires such operations *on demand*, that is, only when a task is actually stolen for real parallelism. Therefore, the overheads of these operations are usually small enough to be ignored. However, sometimes they actually degrade performances, for instance, when divided tasks contain too little amount of actual work. Performance degradation is likely to happen on highly parallel environments.

We propose probabilistic guards, a mechanism for work-stealing frameworks where victim workers are probabilistically guarded from thief workers. We can skew the probabilities of eventual success in repeated uniformly random steal attempts among victim workers to be non-uniform as the "relative" values of the probabilities of success in a single steal attempt among victim workers.

A. *Effects of Probabilistic Guards*

Probabilistic guards aim to reduce the total task division cost by preventing thieves from stealing small tasks. Probabilistic guards do not reduce the cost in many multithreaded languages because it is included even when a spawned thread is not stolen. Cilk [1] and Cilk Plus [9] provide a `SYNCHED` pseudovariable and "reducers," respectively, which enable us to apply probabilistic guards to Cilk and Cilk Plus. It is future work to apply probabilistic guards to systems other than Tascell.

Let g_i be the probability that the task of the i-th victim worker is guarded from a single steal attempt, where $1 \leq i \leq n$ and n is the total number of victim workers. In other words, the probability p_i that the task of the i-th victim worker can be divided and stolen by a single steal attempt satisfies that $p_i = 1 - g_i$. We assume that a thief worker repeats a uniformly random choice of a victim worker until it succeeds in stealing a task. Suppose that g_i is constant while a thief worker is trying to steal a task. Then, we get the probability s_i that the thief worker eventually succeeds in stealing a task from the i-th victim worker, as follows.

$$s_i = \frac{1}{n} p_i + \left(\sum_{j=1}^{n} \frac{1}{n} g_j \right) s_i \tag{1}$$

We use recursiveness in Eq. (1). The right-hand side is the sum of the probability that work stealing succeeds on the first attempt, and the product of the probability that it fails on the first attempt and the probability s_i that it succeeds on the subsequent, unboundedly many attempts. By transforming Eq. (1), we get

$$s_i = \frac{p_i}{\sum_{j=1}^{n} p_j} \tag{2}$$

This indicates that s_i is the ratio of p_i to the sum of $p_j (1 \leq j \leq n)$.

The expected value T of how many times a thief worker attempts to steal a task is given by

$$T = \frac{n}{\sum_{i=1}^{n} p_i} = \frac{1}{\overline{p}} \tag{3}$$

where $\overline{p}$ is defined as the probability that a thief worker successfully steal a task at a single attempt. This indicates that T is the inverse of the average of $p_i (1 \leq i \leq n)$.

Every worker has a probabilistic guard. By giving an appropriate probability p_i to each worker, we think that the overheads of small task division can be reduced. If a worker has a large task, we give high p_i for beneficial concurrency. On the contrary, if a worker has a small task, we give low p_i so that the total task division cost will be suppressed.

When we parallelize irregular applications, obviously it is hard to determine the accurate size of a certain task. In that case, we need to estimate approximate sizes of tasks, which cannot be done automatically by work-stealing frameworks. Therefore, we extended the Tascell framework so that users can define the probability that a task of a certain victim worker can be stolen by a single steal attempt. When a task request occurs, users of the Tascell framework, by considering characteristics of the parallelized applications, can tell the framework whether it should spawn a new task or not based on the probability. If permitted, the victim worker divides its task and spawns a new task as usual. Otherwise, it refuses to divide its task. Then, the thief worker repeats steal attempts until it succeeds.

Note that whether a certain task can be stolen or not is probabilistically determined, not absolutely determined by thresholds. That is, work stealing does not completely stop even if the estimated size of a task becomes lower than some constant size; it only becomes less likely to be stolen. We believe that probabilistic guards reduce the total task division overhead without loss of potential concurrency. Moreover, probabilities can be regarded as priorities of tasks to steal when thief workers choose victim workers, which cannot be realized by thresholds. Note that probabilistic guards can emulate thresholds' behavior by setting binary probabilities, 0 and 1.

B. *Upper Limit*

As mentioned above, a thief worker repeatedly tries to steal a task from victim workers while it fails to steal. In theory, a thief may repeat an unbounded number of probabilistically prevented steal attempts if a victim employs a probabilistic guard that rejects steal attempts with a non-zero probability.

A thief prevented from stealing a task probabilistically sleeps for a little time, then it tries to steal a task from victims again. Thereby, the idle time of the thief becomes long and the parallel efficiency of work-stealing may degrade if the steal attempts are repeated a large number of times.

Therefore, we propose a mechanism that invalidates probabilistic guards on demand by setting an upper limit to the number of repeated, probabilistically prevented steal attempts. This mechanism can be described in more detail as follows.

1) A worker becomes idle; that is, it becomes a thief. It keeps count of the number of steal attempts prevented by probabilistic guards since it becomes idle.
2) It tries to steal a task from a randomly-selected victim worker. Since the victim may be guarded from the thief probabilistically, the thief repeats probabilistically prevented steal attempts until it succeeds or the count of steal attempts prevented by probabilistic guards reaches the upper limit.
3) If the count reaches the upper limit, the thief forcibly steals a task from victims. That is, it can invalide probabilistic guards.

C. Extension of Tascell Framework

We extended the Tascell language and introduced a worker-local variable to support probabilistic guards. Users can set a probability that tasks of a worker can be stolen, by using the following statement either in a `worker` function or in a `task_exec` body:

`WDATA.probability =` exp_p;

This assignment statement sets the value of the expression exp_p as the probability for the worker executing this statement. The type of exp_p should be `double`, and its value p should be between `0.0` and `1.0`. For example, the worker is not guarded if p is `1.0`, and the worker is fully guarded if p is `0.0`.

The probability of each worker remains the same unless one of the following events happens:

- A new probability is set by users.
- The worker steals a new task. In this case, the probability is reset to 1 by the Tascell framework.
- The worker sends the result of a task. In this case, the probability is set to the previous value before the worker stole the task.

Note that probabilities are not dynamically calculated when attempts of work stealing occur. Instead, they must be properly set in advance by user programs. We also considered an implementation in which probabilities can be set by users right after the backtracking occurs and the oldest task-spawnable state is restored. However, a preliminary evaluation suggested that the cost of the backtracking operations is too high to examine a probability if work stealing is guarded. Therefore, we decided to let users set probabilities freely in the user programs.

Users need to set the probabilities with proper timing. As we discussed in the previous section, it is typically when the size of the remaining task of a worker changes. Nevertheless, because the actual size of a stolen task is determined after the backtracking occurs and the oldest task-spawnable state is restored, it is inefficient to set probabilities every time the size of the task changes. In our evaluation programs, we set probabilities with the following timing:

- Right after `task_exec` is called.
- During executing $stat_{put}$ in the task request handlers.

We determine a probability by calculating the approximate size of a task which will be stolen next time the work stealing occurs. In addition, we avoid the overhead of frequent recalculations of probabilities. We show the actual parallel implementations of the Barnes-Hut algorithm in the next section as examples.

IV. Parallelizaton of the Barnes-Hut Algorithm

We implemented parallel N-body simulation programs based on *treecode* version 1.4 [10], a serial N-body simulation program. Its implementation can be broken down as follows.

1) Load all bodies one by one and construct the tree.
2) Compute gravitational forces for all bodies and update their potentials and accelerations.
3) Advance one time step and update the velocities and positions of all bodies.

Treecode simulates motions of body clusters by repeating these phases a specified number of times, or until a specified amount of simulation time has elapsed.

In general, parallelizing tree construction of the Barnes-Hut algorithm requires careful implementations in order to achieve ideal performance gain. In the Barnes-Hut algorithm, a single tree is constructed by inserting bodies and dividing cells. Therefore, typical parallelization methods of the Barnes-Hut tree construction are either

1) to make all workers construct the same single tree cooperatively by using mutual exclusion primitives, or
2) to effectively partition constructing parts of the tree among workers, and later merge them all if necessary.

In fact, several tree construction algorithms for parallelization are proposed, as we describe in Section VI. In this study, we implemented two different parallelization algorithms for tree construction, employing probabilistic guards which are expected to improve (preserve) their performance by preventing thieves from stealing small tasks so as to reduce the total task division cost. We call the two algorithms "merging" and "bin." We show the details of these two algorithms below.

A. Tree Construction: Merging Algorithm

1) Base Implementation: The merging algorithm makes workers construct their own local trees from a subset of all bodies, and later merges all the local trees into one global tree. Because each worker constructs its respective, independent tree, the merging algorithm requires no mutual exclusion. However, the merging algorithm requires additional merge operations which do not exist in the serial algorithm.

In order to quickly complete tree merging, it is important to consider how we divide the whole set of bodies to assign to workers. If we make subsets of bodies which are uniformly distributed in the whole space, workers take a long time to merge trees they construct deep tree traversals are required. By contrast, if we make subsets of bodies which are clustered and spatially separated from each other, workers only have to examine nodes around the roots of the trees when merging. We use the old tree constructed at the previous time step to create subsets of such clustered bodies. This is because the Barnes-Hut trees represent the spatial structure of bodies, and we can easily find neighboring bodies by traversing them. The previously constructed tree is not structurally accurate since bodies have moved during force calculation of the previous step. However, we think that it still suffices for our propose.

2) Implementation with Probabilistic Guard: Obviously, a potential cause of performance degradation in this algorithm is the merging operation, which is not included in the serial tree construction algorithm. The amount of the merging overheads also depends partly on how tasks are divided. That is, if a spawned task includes a very small number of bodies, the execution time for merging trees becomes relatively long, compared to the time for inserting bodies.

Therefore, we applied probabilistic guards to reduce the overheads. We define the success probability of a steal attempt as

$$p = min(1, 2P\frac{b_{\text{in}}}{b_{\text{all}}}) \tag{4}$$

where P is the number of all workers, b_{in} is the number of bodies included in the tree structure of the current task, and b_{all} is the total number of all bodies. This definition reflects the ratio of the amount of remaining work each victim worker has. By making workers steal a sufficient number of bodies as much as possible probabilistically, we expect that small tasks are less prone to be spawn, and consequently the number of spawned tasks is reduced.

B. Tree Construction: Bin Algorithm

1) Base Implementation: The bin algorithm makes workers construct a single tree with the usual work stealing, but avoids mutual exclusion by partitioning bodies into *spatially separated* subsets of bodies before "insertion." The bin algorithm is a simple tree-recursive algorithm except that it has the partitioning phase before parallel recursion (like "quick sort" and recursive "bin sort"). At each recursive level, it employs 8 bins to collect partitioned bodies for 8 subnodes, and then each subnode is either empty or nonempty; the nonempty subnode is either a single body or a cell which will be constructed recursively with the corresponding bin of bodies.

We implement each bin as a linked list of arrays of bodies. In order to concurrently partition and collect bodies *without* mutual exclusion, the bin algorithm is required to use *multiple bins* for each subnode according to the concurrency. Fortunately, Tascell features on-demand concurrency and promotes the long-term use of a single bin [8]; that is, a new empty bin is created only for each stolen task, which will be concatenated to the original bin after the stolen task is completed. Therefore, we can use a relatively long array to efficiently collect a relatively large number of bodies before the next array is allocated as a new element of the linked list.

2) Implementation with Probabilistic Guard: Since the bin algorithm is a less redundant algorithm and uses quicker list concatenation operations than tree merge of the merging algorithm, applying probabilistic guards to the bin algorithm would be (not harmful but) less effective than applying probabilistic guards to the merging algorithm.

We applied probabilistic guards in the same manner as the implementation of the merging algorithm; that is, we define the success probability of a steal attempt as in Eq. (4).

C. Force Calculation

In force calculation, Treecode uses a fast algorithm [11]. This algorithm reduces the tree traversing cost by building, for each body, an "interaction list," a list of all bodies and cells which exert on it. Bodies which are close to each other in the space tend to have a similar set of elements in their interaction lists. The fast algorithm of force calculation performs a single tree traversal about all bodies; the algorithm maintains a single interaction list, and sequentially updates it. The forces exerted on a body are calculated using its interaction list when the traverse reaches a leaf node, that is, a body. With this method, bodies which are close to each other in the space can share most of a single interaction list.

```
void gravcalc() {
    list A = a list that has the root node as its element;
    list I = an empty list;
    node p = the root node;
    walktree(A, I, p);
}

void walktree(list A, list I, node p) {
  list nextA = an empty list
                 (has the end of list A as head pointer);
    for (each node a in A) {
        if (a is a cell) {
            calculate the distance of a and p;
            if (the distance of a and p is enough far)
                add a to I;
            else
                add all child nodes of a to nextA;
        } else if (a is not p) {
            add a to I;
        }
    }
    if (nextA has no elements) // p is a body
        calculate forces that each node in I exerts on p;
    else
        walksub(nextA, I, p);
}

void walksub(list A, list I, node p) {
    if(p is a cell)
        for(each child node q of p)
            walktree(A, I, q);
    else
        walktree(A, I, p);
}
```

Fig. 1. Pseudo code of force calculation in the fast algorithm

Pseudo code of force calculation in the fast algorithm is shown in Figure 1. The force calculation begins at `gravcalc`. The force calculation is performed through mutually recursive calls of `walktree` and `walksub`. The list `I` is an interaction list computed until the traverse reaches from root node to node `p`. The list `A` is a list of nodes which have not finished traversing about `p`. The list `nextA`, which is initially empty, is a list to record elements that are newly added next to a tail pointer of list `A`. The array to record elements is shared with list `A`. `Walktree` calculates forces exerted on all of bodies which are included in node `p`. `Walktree` calculates the distance between each element `a` of list `A` and `p`to add `a` to `I` or add child nodes of `a` to `nextA`. That is, based on the distance, `walktree` determines whether the expansion of child nodes of `a` is need or not. Because `p` is always a body if no elements have not been added to `nextA`, `walktree` calculates forces exerted on `p` with the interaction list at that time. When any elements have been added to `nextA`, `walktree` calls `walksub` and calculates forces exert on child nodes of `p` if `p` is a cell.

We parallelized force calculation by modifying `walksub`. If `p` is a cell, `walksub` calls `walktree` for each child node `q` of `p` through `for` statement. By using parallel `for` statement in Tascell instead of the `for` statement and we implemented parallelization of force calculation.

To calculate forces in parallel, the interaction list is copied

TABLE I
EVALUATION ENVIRONMENT.

	Linux PC Server
CPU	AMD Opetron 6366 HE 1.6GHz 16-Core * 2
Memory	32GB
OS	Linux 2.6.32(x86_64)
Compiler	Tascell Compiler with `-O2` optimizer (The force calculation is optimized with `-O3`.)

every time tasks are divided. Because the cost of copying interaction lists is considerably large, dividing small tasks causes performance degradation. Therefore, applying the probabilistic guards to the force calculation is important to reduce the total task division cost. We define the success probability of a steal attempt as

$$p = min(1, 2P\frac{m_{\text{sub}}}{m_{\text{all}}}) \tag{5}$$

where P is the number of workers, m_{sub} is the sum of the masses of bodies to update and m_{all} is the total mass of all the bodies.

V. PRELIMINARY EVALUATIONS

In this section, we present the preliminary evaluation results of parallel implementations of the Barnes-Hut algorithm. The evaluation environment of preliminary evaluatin is shown in Table I.

A. *Performance evaluation of Probabilistic Guards*

First, we present the performance evaluation results of parallel implementations of the Barnes-Hut algorithm. In addition to the two tree construction algorithms shown in Section IV, we implemented parallelied force calculation on a Barnes-Hut tree also in Tascell for evaluation. In each algorithm, we compared effectiveness of probabilistic guards to that of threshold methods. We implemented thresholds by giving binary probabilities as we mentioned in Section III.

The evaluation results are shown in Figures 2 to 6. Explanation and legends in the figures are summarized as follows. We considered a dispersion of execution results caused by random numbers generation in probabilistic guard and measured execution time with 22 random number seeds. Then we showed first quartile and third quartile as error bars in Figures 2 to 6.

- Figure 2 shows the execution times of constructing a Barnes-Hut tree from n bodies. In this preliminary evaluation, we adopted 1000000 as n and compared the merging algorithm and the bin algorithm.
  - **bin-original** shows execution times of the bin algorithm without probabilistic guards.
  - **bin-pg** shows execution times of the bin algorithm with probabilistic guards. Its probability $p = min(1, kP\frac{b_{\text{in}}}{b_{\text{all}}})$ where P is the number of workers, the coefficient k is 2 in this preliminary evaluation, b_{in} is the number of bodies to insert and b_{all} the number of all of bodies.
  - **bin-threshold** shows execution times of the bin algorithm with probabilistic guards based on thresholds. $p = 1$ if $kP\frac{b_{\text{in}}}{b_{all}} > 1$ where k is 24, $p = 0$ otherwise.
  - **merge-original** shows execution times of the merging algorithm without probabilistic guards.
  - **merge-pg and merge-threshold** show execution times of the merging algorithm with probabilistic guards. p's are defined in the same manner as **bin-pg** and **bin-threshold**, respectively. In **merge-pg**, k is the same as **bin-pg** (that is, $k = 2$), but k is 3 in **merge-threshold**.
- Figure 3 shows parallel efficiency in force calculation. The vertical axis, **efficiency** is calculated with $t_1/t_P/P$ where t_1 ranges over execution times with one worker without probabilistic guards and t_P ranges over execution times in parallel.
  - **non-pg** shows parallel efficiency in force calculation without probabilistic guards.
  - **pg** shows parallel efficiency in force calculation with probabilistic guards. $p = min(1, kP\frac{m_{\text{sub}}}{m_{\text{all}}})$ where k is 2, m_{sub} is the sum of the masses of bodies to update, and m_{all} is the total mass of all the bodies.
  - **threshold** shows parallel efficiency in force calculation with probabilistic guards based on thresholds. $p = 1$ if $kP\frac{m_{\text{sub}}}{m_{\text{all}}} > 1$ where k is 16, $p = 0$ otherwise.
- Figures 4 to 6 show the relative speedups of the merging algorithm, the bin algorithm and force calculation, respectively. The speedup values are performance ratios for computation times of **non-pg** executed with one worker.
  - **non-pg, pg and threshold** in Figure 4, 5 and 6 are same as **merge-original, merge-pg and merge-threshold** in Figure 2, **bin-original, bin-pg and bin-threshold** in Figure 2 and **non-pg, pg and threshold** in Figure 6, respectively.

Let us consider how to set the parameter k in the probability expression of steal success. Let T be the total amount of work to be done in parallel execution. Then, if T is divided equally by P workers, each worker can have work of size T/P equally. Let T_i be the amount of work which the i-th worker currently has. An ideal way is to statically divide T so that $T_i = T/P$ at the beginning of parallel execution, but we had better adopt dynamic load balancing, since the ideal way is practically difficult. Although T_i changes by progress of time or dynamic load balancing, we should divide T_i with work stealing if we estimate $T_i > T/P$, because the i-th worker has the larger work than T/P. On the other hand, if T_i is smaller than $T/P/100$, work stealing can reduce load imbalance by only about 1%. The effect of this work stealing is minute compared to its overheads. Therefore, when thresholds determine whether steal attempts succeed or not by judging $T_i > T/P/k$ (that is, $kP(T_i/T) > 1$), defining k as 10 to 20 would be effective to reduce total load imbalance to 5% to 10%, while suppressing division overheads. By contrast, when probabilistic guards determine whether steal attempts succeed or not by evaluating $min(1, kP(T_i/T))$ as steal success rates, defining k as 2 makes probabilistic guards *working* when T_i becomes less than or equal to a half of T/P (that is, $T/P/k$) and makes the success rate be $1/2$ when T_i becomes a quarter of T/P (that is, $T/P/k/2$). Moreover, if T_i is $1/8$ of T/P (reducing load imbalance by about 12.5%), the rate decreases to 25%, and we expect that the steal is likely induced to the other victims which have success rates larger than 25%. From the above consideration, we think that setting $k = 20$ for thresholds and $k = 2$ for probabilistic guards is reasonable. In fact, when we conducted a preliminary evaluation of probabilistic guards with various k in some environment, we confirmed the best performance with each value of parameter k mentioned in the evaluation settings.

In Figure 2, we cannot confirm the difference in performance with probabilistic guards and thresholds. In Figure 3, it took 273 seconds to compute with one worker ; force calculation needs a large amount of time compared with contructing a tree. We can confirm that **pg** shows the probabilistic guards suppresses performance degradation of parallel efficiency. In Figure 6, **pg** shows the highest performance with 8 or more workers, and we can also see the effect of the probabilistic guards. As we mentioned in Section IV, the interaction lists which take the relatively large copying cost are copied when dividing a task. For this reason, the performance was improved because probabilistic guards prevented thief workers from stealing small tasks and reduced the total cost of copying the interaction lists.

On the other hand, the performances of the program with probabilistic guards are totally worse than the program without it in Figures 4 and 5. Because probabilistic guards reduce the total task division cost by preventing stealing small tasks probabilistically, if the original implementation requires only small amount of copying or merging costs when dividing a task, the loss of the work-stealing chances cause performance degradation.

B. Performance Analyzing of Probabilistic Guards

In performance evaluation, we confirmed the efficacy of probabilistic guards and good performance in force calculation. However, as we mentioned in Section III, probabilistic guards reject steal attempts with a non-zero probability, and a thief may repeat an unbounded number of probabilistically prevented steal attempts. Therefore, firstly, we counted the number of times a thief is probabilistically prevented from stealing attempts until success during an parallel execution of the Barnes-Hut algorithm. Secondly, we applied various upper limits to probabilistic guards and examined their impact. Note that the number of workers in parallel execution of the Barnes-Hut algorithm is fixed to 16. As with above-mentioned, we measured execution time with 22 random number seeds, then we showed first quartile and third quartile as error bars in Figures 8 and 9.

The evaluation results are shown in Figures 7 to 9. Figure 7 shows the cumulative relative frequency distribution of the

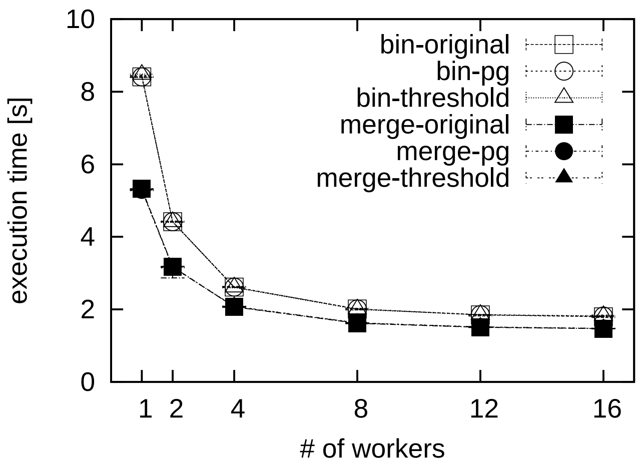

Fig. 2. Execution times of the merging algorithm and bin algorithm.

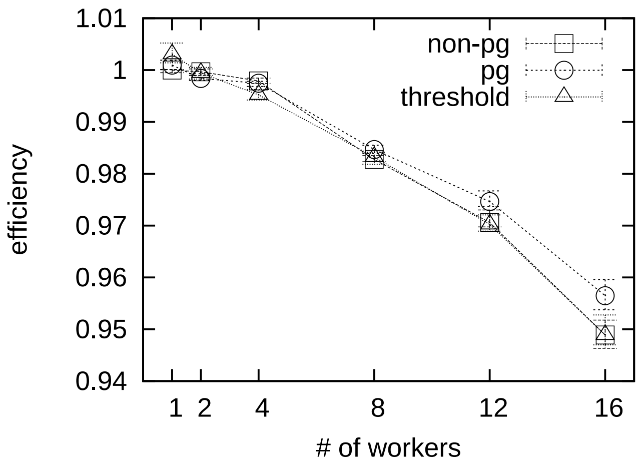

Fig. 3. Parallel efficiency in force calculation.

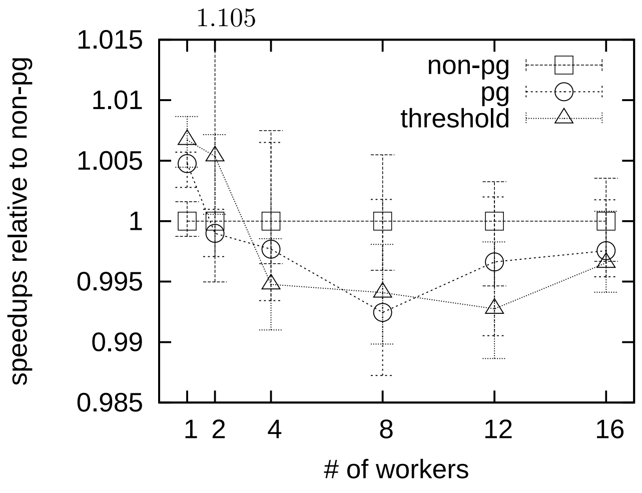

Fig. 4. The effect of probabilistic guards in the merging algorithm.

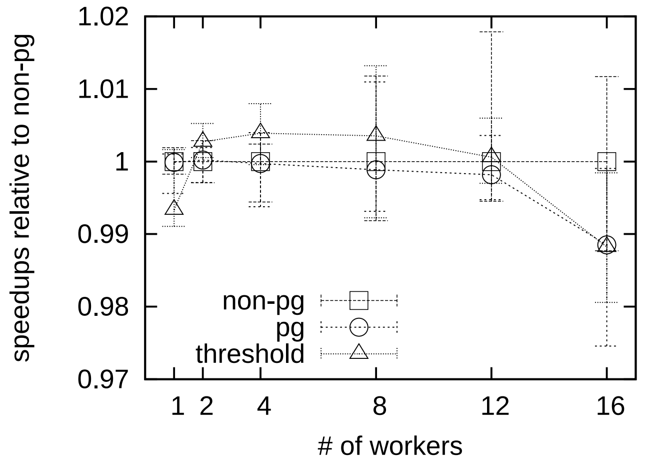

Fig. 5. The effect of probabilistic guards in the bin algorithm.

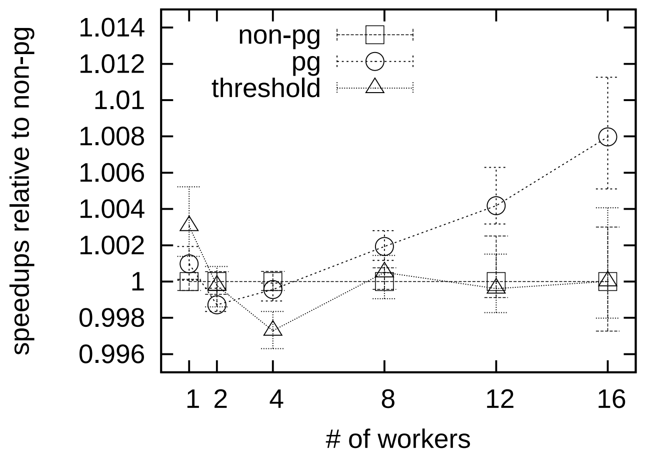

Fig. 6. The effect of probabilistic guards in force calculation.

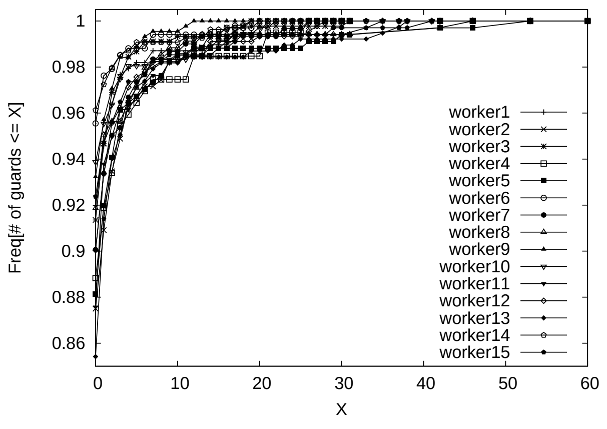

Fig. 7. The cumulative relative frequency distribution of the number of repeated, probabilistically prevented steal attempts

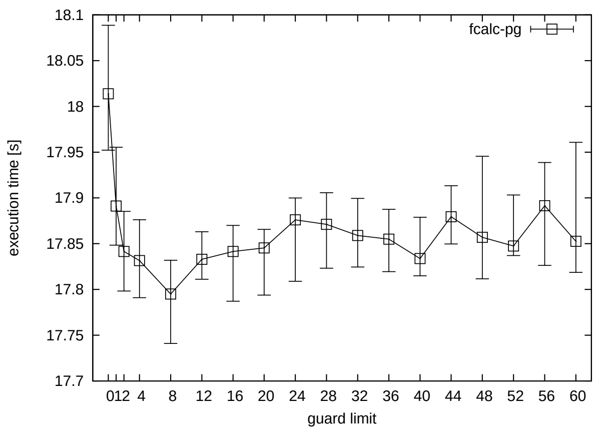

Fig. 8. Execution times of the force calculation for various upper limits.

number of repeated, probabilistically prevented attempts in parallel execution of Barnes-Hut algorithm. For the horizontal axis X, the vertical axis shows the frequency that the number of times that a thief have been prevented from stealing a task probabilistically until success was X or less. With this result, if we set an upper limit for probabilistic guards to about 60, the upper limit is useful for the situation a thief repeat an unbounded number of probabilistically prevented steal attempts without loss of the probabilistic guard function.

Figures 8 and 9 show execution times of the force calculation for various upper limits and the number of tasks copied in the calculation for various upper limits, respectively. In Figure 8, we can see that execution time becomes shorter as the upper limit increases from 0 to 8 because reducing the total cost of copying interaction lists have relatively large copying cost. In addition, we can also see the highest performance when the upper limit is 8. When we set the upper limit to half of the number of all workers, a thief gets a chance of steal attempts for half of the number of all workers too. By setting the upper limit in this manner, the thief can steal a task which is large enough to exceed the overheads of dividing tasks, and we think it leads to good performance.

VI. Related Work

Several implementations of work stealing have been proposed [1], [4], [12], [13], [2], [14]. These work stealing frameworks are usually realized as multithreaded languages or their variants implemented as libraries. Unlike multithreaded languages, Tascell [5] provides logical-thread-free on demand concurrency; an idle worker requests a task, and a new task is spawn only when requested.

Leiserson *et al.* analyzed the setting of work stealing in multithreaded computations and established tight upper bounds on the number of steals when the computation can be modeled by rooted trees [15]. They analyzed the setting of work stealing on the presupposition that steal attempts always succeed against our presupposition that the steal attempts can be rejected probabilistically.

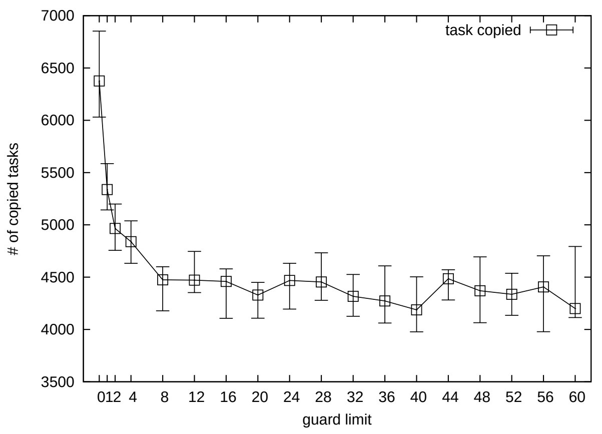

Fig. 9. The number of tasks copied in the force calculation for various upper limits.

A message passing implementation [12] of LTC employs a polling method as in Tascell where the victim detects a task request sent by the thief and returns a new task (the continuation of a previously spawned logical thread) created by splitting the present running task. StackThreads/MP [14] also employs this technique. Polling methods often improve performance by avoiding "memory barrier" instructions. Cilk [1] and TBB [13] employ deques which require "memory barrier" instructions. Since "memory barrier" instructions are generally heavy on modern parallel architectures, thresholds (or cutoffs) are often required in these systems. The Tascell framework basically does not require thresholds for mitigating the cost of memory barrier instructions. Thus, probabilistic guards are useful for reducing average concurrency without loss of potential concurrency.

Singh *et al.* provided several lock-based parallel tree construction algorithms for the Barnes-Hut N-body simulation, contributing to the SPLASH benchmarks [16]. Although their initial implementation was based on a lock-based parallel tree construction algorithm, they assigned a worker a task for inserting bodies that are spatially separated from those of the other workers so that the mutual exclusion overheads can be reduced. However, their evaluation showed that the tree construction accounted for a greater ratio of the total computation time with an increasing number of processors, suggesting that it can be a potential performance bottleneck when using a large number of processors. They also proposed another parallel tree construction algorithm in which each processor constructed its own separated tree and merged it with the shared tree. The latter algorithm exhibited better performance than the former algorithm. Note that all these algorithms are based on static load balancing. They later proposed a space-based approach [17], which is also based on static load balancing.

Many researchers have studied large-scale N-body simulations using high-performance computing clusters. Current studies employ general-purpose graphics processing units (GPGPUs) to accelerate intensive floating-point operations such as the calculation of gravitational forces. However, tree construction is typically performed on CPUs [18], [19] because of its numerous branch instructions and irregular memory access patterns, which are unsuitable for GPGPU processing. Parallel implementations of tree construction described in this paper are suited for many-core CPUs, which will be commonly used in the near future. We also expect our proposal to be useful in many-core nodes in large-scale computing environments.

VII. Conclusion

In this study, we extended an existing work-stealing framework Tascell with probabilistic guards, a new mechanism by which victim worker's tasks are probabilistically guarded from thief workers. Through a preliminary evaluation of the Barnes-Hut algorithm with probabilistic guards, we confirmed their effectiveness. We also examined an upper limit to the number of repeated, probabilistically prevented steal attempts until success. Through a preliminary evaluation of numbers of prevented steal attempts, we confirmed that reasonable upper limits can be introduced while keeping essential effects of probabilistic guards. Finally, we conducted a preliminary evaluation of the performance of probabilistic guards with various upper limits.

Tascell also supports computations in distributed memory environments and there are many interesting research directions in extending Tascell for such environments with probabilistic guards. We will also analyze the effectiveness of our concept in other systems such as Cilk.

Acknowledgement

This work was supported in part by JSPS KAKENHI Grant Numbers 25730041 and 26280023. We thank the anonymous reviewers for their helpful and constructive comments.

References

[1] M. Frigo, C. E. Leiserson, and K. H. Randall, "The Implementation of the Cilk-5 Multithreaded Language," in *Proc. of the ACM SIGPLAN Conf. PLDI*, 1998, pp. 212–223.

[2] IBM Research, "X10: Performance and productivity at scale," http://x10-lang.org/.

[3] R. H. Halstead, Jr., "New Ideas in Parallel Lisp: Language Design, Implementation, and Programming Tools," in *Parallel Lisp: Languages and Systems*, 1990, pp. 2–57.

[4] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr., "Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, Jul. 1991, pp. 264–280.

[5] T. Hiraishi, M. Yasugi, S. Umatani, and T. Yuasa, "Backtracking-based Load Balancing," in *Proc. of the 14th ACM SIGPLAN Symposium PPoPP*, Feb. 2009, pp. 55–64.

[6] H. Yoritaka, K. Matsui, M. Yasugi, and T. Hiraishi, "Proposing probabilistic guards for a work-stealing framework and their performance analysis," in *the 17th JSSST Workshop on Programming and Programming Languages*, Mar. 2015, (In Japanese).

[7] J. Barnes and P. Hut, "A hierarchical $O(N \log N)$ force-calculation algorithm," *Nature*, vol. 324, pp. 446–449, Dec. 1986.

[8] M. Yasugi, T. Hiraishi, S. Umatani, and T. Yuasa, "Parallel graph traversals using work-stealing frameworks for many-core platforms," *Journal of Information Processing*, vol. 20, no. 1, pp. 128–139, Jan. 2012.

[9] Intel Corporation, "Intel Cilk Plus," http://software.intel.com/en-us/intel-cilk-plus.

[10] J. E. Barnes, "Treecode Guide," http://www.ifa.hawaii.edu/ barnes/treecode/treeguide.html.

[11] ——, "A Modified Tree Code: Don't Laugh; It Runs," *Journal of Computational Physics*, vol. 87, pp. 161–170, 1990.

[12] M. Feeley, "A message passing implementation of lazy task creation," in *Proc. of the Intl. Workshop on Parallel Symbolic Computing: Languages, Systems, and Applications*, ser. Lecture Notes in Computer Science, no. 748. Springer-Verlag, 1993, pp. 94–107.

[13] Intel Corporation, *Intel Threading Building Block Reference Manual*, 2007, http://threadingbuildingblocks.org/.

[14] K. Taura, K. Tabata, and A. Yonezawa, "StackThreads/MP: Integrating futures into calling standards," in *Proc. of ACM SIGPLAN Symposium PPoPP*, May 1999, pp. 60–71.

[15] C. E. Leiserson, T. B. Schardl, and W. Suksompong, "Upper bounds on number of steals in rooted trees," *Theory of Computing Systems*, pp. 1–18, Feb. 2015.

[16] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. L. Hennessy, "Load Balancing and Data Locality in Adaptive Hierarchical N-body Methods: Barnes-Hut, Fast Multipole, and Radiosity," *Journal of Parallel and Distributed Computing*, vol. 27, pp. 118–141, June 1995.

[17] H. Shan and J. P. Singh, "Parallel tree building on a range of shared address space multiprocessors: Algorithms and application performance," in *Proc. of the 12th IPPS/SPDP*, 1998, pp. 475–484.

[18] T. Hamada and K. Nitadori, "190 TFlops Astrophysical N-body Simulation on a Cluster of GPUs," in *Proc. of the ACM/IEEE Intl. Conf. SC10*, 2010.

[19] P. Jetley, L. Wesolowski, F. Gioachin, L. V. Kalé, and T. R. Quinn, "Scaling Hierarchical N-body Simulations on GPU Clusters," in *Proc. of the ACM/IEEE Intl. Conf. SC10*, 2010.