Regular Paper

Work-stealing Strategies That Consider Work Amount and Hierarchy

Ryusuke Nakashima[1,a)] Masahiro Yasugi[2,b)] Hiroshi Yoritaka[1,†1]
Tasuku Hiraishi[3] Seiji Umatani[4]

Received: December 11, 2020, Accepted: March 12, 2021

Abstract: This paper proposes work-stealing strategies for an idle worker (thief) to select a victim worker. These strategies avoid small tasks being stolen to reduce the total task-division cost. We implemented these strategies on a work-stealing framework called Tascell. First, we propose new types of priority- and weight-based steal strategies. Programmers can let each worker estimate and declare, as a real number, the amount of remaining work required to complete its current task so that declared values are used as "priorities" or "weights". With a priority-based strategy, a thief selects the victim that has the highest known priority at that time. With a weight-based non-uniformly random strategy, a thief uses the relative weights of victim candidates as their selection probabilities. Second, we propose work-stealing strategies to alleviate excessive intra-node work stealing and excessive "steal backs" (or leapfroggings); for example, we allow workers to steal tasks from external nodes with some frequency even if work remains inside the current node. Our evaluation uses a parallel implementation of the "highly serial" version of the Barnes-Hut force-calculation algorithm in a shared memory environment and five benchmark programs in a distributed memory environment.

Keywords: parallel programming languages, work stealing, priority, weight, concurrency, many-core, Barnes-Hut algorithm

1. Introduction

With the proliferation of parallel-computing environments including multicore processors, mechanisms and techniques that enable efficient parallel computing for a wide range of applications are gaining importance.

Work stealing is a technique for parallelizing applications. An idle worker (thief) steals a task from another worker (victim) to provide efficient dynamic load balancing. Multithreaded languages [1], [2], [3], [4], [5] provide dynamic load balancing based on work-stealing techniques. Cilk [2] renders all workers busy by creating plenty of "logical threads" and adopting the oldest-first work-stealing strategy.

A logical-thread-free parallel programming/execution framework called Tascell implements backtracking-based load balancing [6]. A Tascell worker spawns a real task by temporarily backtracking and restoring its oldest task-spawnable state only when requested by an idle worker. This method eliminates the costs of spawning and managing logical threads. Furthermore, Tascell promotes the long-term (re)use of workspaces, improves the locality of reference, and enables delayed workspace copying.

In the base implementation of Tascell, victims are randomly selected based on uniformly random selection. However, victims with small tasks may be selected, and dividing such tasks may degrade parallel efficiency because task division may involve workspace and data copying.

In contrast to uniformly random selection, we proposed *probabilistic guards* (and *virtual probabilistic guards*) [7], [8] as a non-uniformly random steal strategy. Probabilistic guards can prevent thieves from stealing small tasks from victims probabilistically.

However, using probabilistic guards involves the following issues:

- Only values between 0 and 1 can be used as probabilities (i.e., values exceeding 1 cannot be used).
- For setting a probability, the effect of the guard must be considered carefully.
- Priority-based selection is not supported.

This paper [*1] proposes two new types of priority- and weight-based steal strategies. Tascell programmers can let each worker estimate and declare as a real number the amount of remaining work required to complete its current task so that declared values are used as priorities or weights in the enhanced Tascell framework. To reduce the total task-division cost, the proposed strategies avoid stealing small tasks. These strategies offer the following advantages:

[1] Graduate School of Computer Science and Systems Engineering, Kyushu Institute of Technology, Iizuka, Fukuoka 820–8502, Japan
[2] Department of Computer Science and Networks, Kyushu Institute of Technology, Iizuka, Fukuoka 820–8502, Japan
[3] Academic Center for Computing and Media Studies, Kyoto University, Kyoto 606–8501, Japan
[4] Department of Information Sciences, Faculty of Science, Kanagawa University, Hiratsuka, Kanagawa 259–1293, Japan
[†1] Presently with Presently with Ad-Sol Nissin Corporation
[a)] ryusuke@pl.ai.kyutech.ac.jp
[b)] yasugi@ai.kyutech.ac.jp

[*1] Part of our proposals can also be found in Ref. [9].

- A value exceeding 1 can be used to achieve further performance improvements.
- Using a value larger than 1, the estimated amount of remaining work can be simply declared, which is much easier than setting a probability value.
- Priority-based selection is essentially different from (non-uniformly) random selection.

For distributed memory environments, in the conventional work stealing strategies of Tascell, only a representative worker sends a task request to external nodes only when there are no workers in the same node that can accept a task request. This *hierarchical* strategy is expected to reduce the number of inter-node work steals. However, even if larger tasks are available in external nodes, this can cause excessive intra-node work stealing; that is, intra-node workers can divide tasks into a large number of small tasks.

Furthermore, in the conventional work stealing strategies of Tascell, a worker that waits for the result of a stolen task sends a task request only to the worker that stole the task ("steal backs", or "leapfroggings" [10]). This constrained strategy guarantees the maximum size of execution stacks. However, this can cause excessive mutual "steal backs" of small tasks among involved workers.

To alleviate excessive work stealing among intra-node workers, this paper proposes another work-stealing strategy in which any worker can send a task request to external nodes periodically even if some workers in the same node can accept a task request.

To alleviate excessive mutual steal backs, this paper proposes to relax the constraint to some degree; in the relaxed work-stealing strategy, a worker that waits for the result of a stolen task can send non-"steal back" task requests within a certain number of times.

We applied the proposed priority- and weight-based victim selection to the Barnes–Hut algorithm [11]. The Barnes–Hut $O(N \log N)$ tree algorithm is widely used for N-body simulations. In Tascell, Treecode [12] (a fast "highly serial" algorithm [13]) is well parallelized [7], [8].

The contributions of this paper are as follows:

- We propose new types of work-stealing strategies based on priority- and weight-based selection.
- We propose two more work-stealing strategies to alleviate excessive intra-node work stealing and excessive mutual "steal backs".
- We extended the Tascell framework and implemented the four proposed strategies.
- Based on a parallelized implementation [7], [8] of the Barnes–Hut algorithm [11], [12], [13], we present a simplified parallelized implementation for priority- and weight-based selection.
- We present the evaluation results of the parallelized Tascell programs. With 500,000 bodies and 96 workers, performance improvements of 20.3% (24.5%) and 13.8% for priority- ("steal back" constraint-relaxed, priority-) and weight-based selection, respectively, are achieved over uniformly random selection. With 1,000,000 bodies and 96 workers, performance improvements of 10.0% (19.4%) and 6.4% for priority- ("steal back" constraint-relaxed, priority-) and weight-based selection, respectively, are achieved over uniformly random selection.
- We present the performance evaluation in a distributed memory environment with five Tascell programs using the four proposed strategies and their combinations.

This paper is organized as follows. First, we provide a brief description of the Tascell framework in Section 2. We explain conventional work-stealing strategies (i.e., (virtual) probabilistic guards) in Section 3. In Section 4, we propose new types of priority- and weight-based selection. In Section 4, we also propose to alleviate excessive intra-node work stealing and excessive mutual "steal backs". In Section 5, we discuss parallel implementations of the Barnes–Hut algorithm. In Section 6, we present the evaluation results.

2. Tascell Framework

The Tascell framework [6] is a programming and execution framework for an extended C language called the Tascell language.

In Tascell, computations are accomplished by 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 specified as its `task_exec` body. When a worker receives a task, it invokes the associated function to complete its work. In Tascell, an idle worker (thief) can request a task from a loaded worker (victim). When receiving a task request, the victim creates a new task by dividing its own task and returns it to the thief. Subsequently, the thief performs the received task and returns its result to the victim.

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) spawns a task (and changes the execution path to receive the result of the task),
(3) returns from the backtracking, and
(4) resumes its own task.

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

For performing a temporary backtracking, we employ mechanisms for legitimate execution stack access [14], [15], [16], such as "L-closures" and "closures", with which a running program/process can legitimately access data deeply in execution stacks (C stacks).

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. In fact, Tascell outperformed Cilk and Cilk Plus, as shown in Refs. [6] and [17], respectively. Moreover, we can parallelize some "highly serial" applications in a straightforward manner, in which a worker continuously and serially updates a single workspace; this is because Tascell exhibits

```
int a[12];    // manage unused pieces
int b[70];    // the board, with (6+sentinel) × 10 cells
// Try from the j0-th piece to the 12th piece in a[].
// The i-th piece for i<j0 is already used.
// b[k] is the first empty cell in the board.
int search (int k, int j0)
{
  int s=0; // the number of solutions
  for (int p=j0; p<12; p++) {  // iterate through unused pieces
    int ap=a[p];
    for (each possible direction d of the piece) {
      ... local variable definitions here ...
      if (Can the ap-th piece in the d-th direction be placed on the board b?);
      else continue;
      Set the ap-th piece onto the board b and update a.
      kk = the next empty cell;
      if (no empty cell?) s++;          // a solution found
      else s += search (kk, j0+1);  // try the next piece
      Backtrack, i.e., remove the ap-th piece from b and restore a.
    }
  }
  return s;
}
```

Fig. 1 C program (pseudo-code) that performs backtrack search for finding all possible solutions to the Pentomino puzzle.

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 required.

Figure 2 is a parallelized Tascell program that performs backtrack search for finding all possible solutions to the Pentomino puzzle based on the C code shown in **Fig. 1**.

We defined a structure of task objects named `pentomino` in the Tascell program. The fields `k`, `i0`, `i1`, `i2`, `a`, and `b` are declared as the input of a `pentomino` task. The field `s` is declared for storing the result. A Tascell worker that receives a `pentomino` task executes the `pentomino task_exec` body. In the body, the worker can refer to the received task object by the keyword `this`. For example, in `task_exec` in Fig. 2, the worker function `search` is called with the values of the input fields of the task object.

The `search` function divides an iterative computation using a Tascell parallel `for` loop construct. It is syntactically denoted by

`for(int` *identifier* : $expr_{from}$, $expr_{to}$`)` $stat_{body}$
`handles` *task-name* `(int` $identifier_{from}$, `int` $identifier_{to}$`)`
{ $stat_{put}$ $stat_{get}$ }.

When the implicit task-request handler (available during the iterative execution of $stat_{body}$) is invoked, the upper half of the remaining iterations are spawned as a new *task-name* task, whose object is initialized by $stat_{put}$. In $stat_{put}$, the actual assigned range can be referred to by $identifier_{from}$ and $identifier_{to}$. The worker handles (merges) the result of the spawned a task by executing $stat_{get}$. Note that a worker performs iterations for a parallel `for` loop sequentially unless requested.

Parallel `for` statements may be nested dynamically in their $stat_{body}$. Therefore, multiple task-request handlers may be available simultaneously. Each worker attempts to detect a task request by polling at every parallel `for` statement. When detecting a task request, it invokes as old a handler as possible.

```
task pentomino {
  out: int s;    // output
  in: int k, i0, i1, i2;
  in: int a[12]; // manage unused pieces
  in: int b[70]; // the board, with (6+sentinel) × 10 cells
};
task_exec pentomino {
  this.s =
    search (this.k, this.i0, this.i1, this.i2, &this);
}
worker int search (int k, int j0, int j1, int j2,
                   task pentomino *tsk)
{
 int s=0; // the number of solutions
 // parallel for construct in Tascell
 for (int p : j1, j2)
  {
   int ap=tsk->a[p];
   for (each possible direction d of the piece) {
    ... local variable definitions here ...
    if(Can the ap-th piece in the d-th direction be placed on the board
tsk->b?);
    else continue;
    dynamic_wind // construct for specifying undo/redo operations
     { // do/redo operation for dynamic_wind
       Set the ap-th piece onto the board tsk->b and update tsk->a.
     }
     { // body for dynamic_wind
       kk = the next empty cell;
       if (no empty cell?) s++; // a solution found
       else                     // try the next piece
        s += search (kk, j0+1, j0+1, 12, tsk);
     }
     { // undo operation for dynamic_wind
       Backtrack, i.e., remove the ap-th piece from tsk->b and restore tsk->a.
     } // end of dynamic_wind
   }
  }
 handles pentomino (int i1, int i2)
     // Declaration of this and setting a range (i1-i2) is done implicitly
  {
   // put part (performed before sending a task)
   { // put task inputs for upper half iterations
     copy_piece_info (this.a, tsk->a);
     copy_board (this.b, tsk->b);
     this.k=k; this.i0=j0; this.i1=i1; this.i2=i2;
   }
   // get part (performed after receiving the result)
   { s += this.s; }
  } // end of parallel for
 return s;
}
```

Fig. 2 A Tascell program that performs backtrack search for the Pentomino puzzle.

2.1 Task Request

In the conventional work stealing strategy of Tascell, a thief worker executes the following steps to select which victim worker to send a task request.

(1) A thief randomly selects a victim among other workers in the same computing node and sends a task request to it. If the victim can spawn a task, it spawns a task and sends it to the thief, as described above. Otherwise, the victim sends a reject message to the thief.
(2) If the thief receives a task as a response to the request in (1), it executes the task. If it receives a reject message, it selects another worker in the same node as a victim and sends a task request to it.
(3) If the thief receives reject messages from all workers in the same computing node (that is, there are no task-spawnable workers in the node) and the thief is the representative worker in the node, it sends a task request to the *Tascell server*. If the all-rejected thief is not the representative worker in the node, it skips to step (6).

(4) On receiving a task request, the Tascell server randomly selects a computing node from those connected to the server, excluding the request sender, and forwards the request to it.

(5) The computing node that receives this task request from the Tascell server checks the workers in the node in a random order. If the node contains a worker with a task that can be spawned, the worker spawns the task and sends the task to the thief via the Tascell server. Otherwise, the node sends a reject message to the thief, again via the Tascell server.

(6) If the thief cannot acquire a task even from any external computing nodes, it returns to (1) and retries the work-stealing process after a short but gradually long pause period.

2.2 Stealing Back

When a Tascell worker w_1 cannot process its running task without receiving the result of a task t that has been sent to another worker w_2 as part of the running task, the worker w_1 suspends the task t and tries to steal another task as a thief rather than becoming idle waiting for the result. To implement this mechanism, each worker has its own *task stack* to manage tasks assigned to the worker: when a worker steals a task from another worker, it is pushed to the top of the worker's task stack keeping suspended tasks below the top.

When the worker w_1 performs such a work steal caused by a synchronization delay, the victim of the request to steal is not selected using the strategy explained in Section 2.1, but the thief w_1 *steals back* a task from the worker w_2 to which the task t causing the synchronization delay has been assigned.

Using this technique, called *Leapfrogging* [10], we can guarantee that the maximum sizes of the workers' task stacks and execution stacks are at most a constant times as large as those in a "task-creating single worker execution", where a program is executed by a single worker as follows.

- The worker spawns all stealable subtasks and pushes them to its *subtask stack*, although there are no other workers that send task requests. For example, when the worker executes a parallel `for` loop with N iterations, it pushes $\log N$ subtasks to the stack *at first*.
- The worker pops a task from its *subtask stack* to start the task on top of its *task stack*.

3. Probabilistic Guards

In this section, we explain *probabilistic guards* and *virtual probabilistic guards* [7], [8], which we previously proposed to obtain better performance than uniformly random victim selection.

(Only) on stealing, the Tascell framework may need to copy the necessary data for the thief'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 degrades the performance of parallel execution because it increases the total task-division cost. The task-division costs may also include costs to merge the results of divided tasks. As the uniformly random strategy selects victims at random, the frequency of dividing small tasks may be problematic.

Probabilistic guards are aimed at reducing the total task-division cost by preventing thieves from stealing small tasks. Each (victim) worker can declare a probability to accept a steal attempt. The victim who receives a task request decides whether or not the steal attempt is successful based on the probability value set for itself before dividing the task. For a successful steal attempt, the victim divides its own task and spawns a new task for the thief. If the victim rejects the task request, the thief repeats probabilistically prevented steal attempts until successful.

By adjusting the probabilities of success in a single uniformly random steal attempt among victims, we can skew the probabilities of eventual success in repeated uniformly random steal attempts to the *relative* values of the adjusted (non-uniform) probabilities.

Let g_i be the adjusted probability that the task of the i-th victim is guarded against a single steal attempt, where $1 \le i \le n$ and n is the total number of victims. In other words, the probability p_i that the task of the i-th victim can be divided and stolen by a single steal attempt satisfies $p_i = 1 - g_i$. We assume that a thief repeats a uniformly random choice of a victim until it succeeds in stealing a task. Supposing that g_i is constant while a thief is attempting to steal a task, as shown in Ref. [7], we obtain the probability s_i that the thief eventually succeeds in stealing a task from the i-th victim as follows:

$$s_i = \frac{p_i}{\sum_{j=1}^{n} p_j} \qquad (1)$$

That is, s_i is the ratio of p_i to the sum of p_j ($1 \le j \le n$).

To set an appropriate probability value for an application, each worker can roughly estimate the work amount (size) of its current task; if the size is below a certain size, the worker can set a small probability value according to the size. By setting smaller probability values to smaller tasks, the overall parallel performance is expected to improve.

3.1 Virtual Probabilistic Guards

In Eq. (1), s_i is the probability that a thief eventually succeeds in stealing a task from the i-th victim. *Virtual probabilistic guards* [7], [8] act as probabilistic guards without repeating probabilistically prevented steal attempts. That is, the thief randomly selects the i-th victim with probability s_i for a single non-uniformly random forced steal attempt.

In implementing virtual probabilistic guards in a shared memory environment, a thief directly reads the probability values of all victims.

Although such read accesses cause data races, we consider the problem only as an impreciseness in non-uniform random selection among victims.

3.2 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 the tasks of a worker can be stolen by using the following statement:

`WDATA.probability =` exp_p`;`

`WDATA` is a structure unique to each worker and is implicitly defined by the Tascell language. The type of expression 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`.

Users can set the probabilities at appropriate timings: 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 at the following events:

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

3.3 Adjusting Probabilities

Let T be the total amount of work to be performed in parallel execution. If T is divided equally by P workers, each worker can have work of size T/P. Let T_i be the amount of work that 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; however, in practice, it is difficult to divide well.

When probabilistic guards determine whether steal attempts are successful using $min(1, kP(T_i/T))$ as the steal success probabilities, defining k as 2 renders probabilistic guards *effective* when T_i becomes less than half of T/P (that is, $T/P/k$); meanwhile, the success probability becomes 1/2 when T_i is 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 approximately 12.5%), the probability decreases to 25%, and we expect that stealing is indirectly guided to other victims that have success probabilities larger than 25%. Therefore, setting $k = 2$ for probabilistic guards is reasonable.

4. Our Proposal

In this section, we propose two new types of work-stealing strategies based on priority- and weight-based selection. In these strategies, the amount of work is estimated as a real number; the value is assigned by the same statement as in Section 3.2. In this section, we also propose to alleviate excessive intra-node work stealing and excessive mutual "steal backs".

4.1 Priority-based Selection

In the priority-based selection, the real value set in the worker is treated as a priority and used for work stealing. The priority-based work stealing proceeds as follows.

(1) When a worker becomes a thief, κ workers are selected randomly from all victims.

(2) The thief reads the priority of each of κ workers and sends a task request to the worker with the highest priority. Unlike probabilistic guards, the steal attempt will succeed if the worker has a task that can be divided.

By stealing from a worker of high priority, stealing small tasks can be avoided and the total task-division cost can be reduced.

The value of κ, that is, the number of randomly selected victim candidates must be selected from the range $1 \leq \kappa \leq n$ (where n is the number of all victim candidates). When $\kappa = n$, the worker with the highest priority is always selected as the victim. When κ is smaller than n, a victim is selected with some degree of randomness while considering the known priorities. In the performance evaluation in Section 6, we measured performances when $\kappa = 3$ and $\kappa = n$.

4.2 Weight-based Selection

The virtual probabilistic guards described in Section 3.1 is a work-stealing strategy that uses a real number between 0 and 1 as a probability value. In the weight-based selection, this real number can be larger than 1 and used as a weight.

Let w_i be the weight of the i-th worker. If n is the total number of victim candidates, then $1 \leq i \leq n$ and $0 \leq w_i$. Since weights are used instead of probability values in the weight-based selection, the probability s_i that the i-th victim is selected can be obtained by the following equation in the same manner as in Eq. (1).

$$s_i = \frac{w_i}{\sum_{j=1}^{n} w_j} \qquad (2)$$

This means that s_i is the ratio of w_i to the sum of w_j $(1 \leq j \leq n)$. The thief can steal a task from the i-th worker with a single steal attempt, similar to virtual probabilistic guards.

4.3 Advantages of Priority- and Weight-based Selection

For probabilistic guards, in Section 3.3, a probability value is given by $\min(1, kP(T_i/T))$.

By contrast, priority- and weight-based selection only require relatively consistent task size estimation. Therefore, expressions such as $kP(T_i/T)$ are not required. In other words, task size T_i can be simply used as a priority or weight, which is advantageous.

In addition, unlike probabilistic guards with probability 1.0, priority and weight values (which can be more than 1.0) are always meaningful for victim selection.

4.4 Alleviation of Excessive Intra-node Work Stealing

As described in Section 2.1, in the conventional work stealing strategy of Tascell, a worker sends a task request to external nodes only when there are no workers inside the same node that can accept a task request. This strategy is expected to reduce the number of inter-node work steals. However, a thief often obtains a small task within the same node, even if larger tasks are available in external nodes. This can cause the increase in the total number of work steals and performance degradation.

To alleviate such excessive intra-node work stealing, we enhanced the conventional strategy so that a worker periodically sends a task request to external node even if there are workers inside the same node that can accept a task request.

To implement this strategy, we let each worker have its own counter that counts the number of task requests accepted by other workers inside the same node (excluding stealing back requests); a thief worker sends a task request to the Tascell server and resets its counter to 0 if the counter value is E.

4.5 Alleviation of Excessive Steal Backs

As explained in Section 2.2, the stealing back mechanism is necessary for guaranteeing the maximum sizes of workers' task stacks and execution stacks.

On the other hand, this mechanism can cause performance degradation because a worker waiting for the result of another task is restricted to stealing back a task from a specific worker even if larger tasks are available in other workers. When a worker completes a small task stolen back in a short time, it tries to steal back another task. This repetition produces a number of work-steals for small tasks between the two workers. Note that such a situation can occur between workers in different computing nodes. Furthermore, a victim of stealing back can also steal back another worker's small task at the same time. Such a chain of stealing back tasks involving a number of workers across computing nodes significantly degrades the performance.

We alleviated such excessive steal backs by allowing a worker waiting for the result of another task to send a non-stealing back task request, until the number of such task requests reaches a certain upper limit.

We implemented this mechanism by letting each worker have a counter that counts the number of non-stealing back requests sent while the worker is waiting for the results of another task. Using this counter, a worker determines whether it sends a non-stealing back task request or not as follows.

(1) When a worker w_1 sends a task request while waiting for the result of a task assigned to another worker w_2, w_1 sends a non-stealing back task request after incrementing the counter if w_1's counter value c_1 is less than an upper limit τ. Otherwise, it sends a stealing back request to w_2.
(2) When a worker w_3 accepts a non-stealing back task request from w_1, it sends its counter value c_3 to w_1 together with a new task.
(3) When w_1 obtains a task from w_3, it updates its counter value c_1 to c_3 if $c_1 < c_3$, before executing the task.
(4) When w_1 receives a result, it resets its counter value c_1 to that was saved when it started waiting for the result.

A thief that does not wait for a result also updates its counter value with a task it stole as (3) described above. Such "counter inheritance" guarantees that the maximum sizes of the workers' task stacks and execution stacks are at most $(\tau + 1)\times$ a constant times as large as those in a "task-creating single worker execution".

5. Barnes–Hut Algorithm

The Barnes–Hut algorithm is a widely used $O(N \log N)$ algorithm that performs N-body simulations. It performs approximation calculations using the fact that the force exerted by bodies (particles) that are far away is smaller than that by neighboring bodies.

In Refs. [7], [8], we implemented parallel N-body simulation programs based on *Treecode* 1.4 [12], 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 is roughly divided into two phases: tree construction and force calculation. They are repeated one or more times to perform N-body simulations. Because most of the computation time is occupied by force calculation, we do not examine tree construction in this paper.

```
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 tail 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. 3 Pseudo-code for force calculation using the fast algorithm.

In the force calculation, *Treecode* uses a fast "highly serial" algorithm [13]. This algorithm reduces the cost of tree search by the fact that nearby bodies in space have a similar "interaction list." Here, the interaction list is all that affects a certain body. In the fast algorithm, the tree search is performed only once for all the bodies, and during that time, a single interaction list is updated sequentially to calculate the force exerted.

Figure 3 shows the pseudo code that performs fast force calculations. The list initialized here actually consists of a sufficiently large array and pointers to the beginning and end of the array elements in use. The force calculation starts with the function `gravcalc` and is performed by mutually recursively calling the functions `walktree` and `walksub`. List `I` is the interaction list updated until the traversal reaches node `p` from the root node. List `A` is an active list of nodes that have not finished traversing about `p`. List `nextA`, which is initially empty, is a new active list to record elements that are newly added next to the tail pointer of list `A`. The array to record elements is shared with list `A`.

`Walktree` calculates forces exerted on all bodies included in node `p`. It 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, whether a search for a child node of `a` is unnecessary is determined. Because `p` is always a body if no elements have been added to `nextA`, `walktree` calculates the forces exerted on `p` with the interaction list at that time. When any element has been added to `nextA`, `walktree` calls `walksub`, which calculates the forces exerted on the child nodes of `p` if `p` is a cell.

We can parallelize "highly serial" force calculation by modifying `walksub`. If `p` is a cell, `walksub` calls `walktree` for each child node `q` of `p` through a `for` statement. By using the parallel `for` statement in Tascell instead of the `for` statement, we parallelized the force calculation [7], [8].

To calculate forces in parallel, an interaction list (and an active list) is deeply copied as a workspace only when a task is lazily divided. Because the cost of deeply copying interaction lists (and active lists) is considerably large, dividing small tasks degrades the performance.

For (virtual) probabilistic guards, as proposed in Refs. [7], [8], the success probability of a steal attempt is given by

$$p = \min\left(1, 2P\frac{m_{\mathrm{sub}}}{m_{\mathrm{all}}}\right), \tag{3}$$

where P is the number of all workers, m_{sub} is the total mass of non-updated bodies, and m_{all} is the total mass of bodies.

By contrast, priorities and weights are simply declared as follows.

$$w = m_{\mathrm{sub}} \tag{4}$$

6. Evaluations

The evaluation environment is summarized in **Table 1**. Every 57-core Xeon Phi co-processor can serve as a computation node with its own memory in a distributed memory environment. We use just a Xeon Phi co-processor (with 228 hardware threads) as a shared memory environment.

The evaluated implementation provides an option that enables a *retry loop* in a node after the rejected first steal attempt; the first victim candidate can be selected with various strategies. The retry loop corresponds to "all workers" in the node in steps (3) and (5) in Section 2.1. Unlike [7], [8], [9], we enable the retry-loop option for *all* evaluations below.

Furthermore, unlike [9], priority-based selection is properly applied even for task requests from external nodes.

The evaluated implementation with the proposed strategies provides users (Tascell programmers) with new opportunities to improve performance. Users can explore the best use of the opportunities; however, even with simple use of the opportunities, we can see considerable improvements as shown below.

6.1 Evaluations in a Shared Memory Environment

In this section, we present the evaluation results of parallel implementations of the Barnes–Hut N-body algorithm. We adopted 500,000 and 1,000,000 separately as N and set the number of time steps so that the force calculation is performed five times.

Table 1 Evaluation environment.

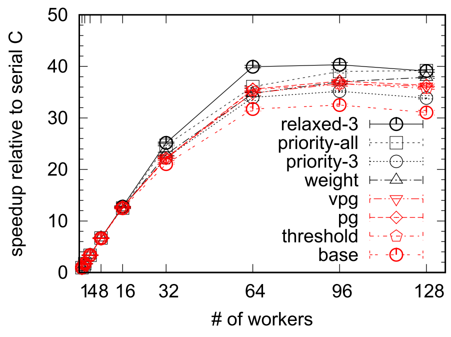
	Xeon Phi
Host processor(s)	Intel Xeon E5-2697 v2 12-core × 2
Host memory	64 GB (shared)
Co-processor(s)	Intel Xeon Phi 3120P 57-core × 4 (four hardware threads per core)
Co-processor Memory	6 GB (for each co-processor)
Network	Each co-processor is connected to the host via PCIe 3.0×16 (Bandwidth = 15.6 GB/s)
OS	CentOS 6.5 (64 bit)
Compiler	Intel Compiler 13.1.3 with -O3 optimizers
Closure	Trampoline-based implementation (compatible with the GCC extension [18])
Tascell server	Steel Bank Common Lisp 1.2.7 (runs on the host processors)

We applied priority- and weight-based selection to the parallel implementation of *Treecode* and compared speedups relative to serial C. In addition, we applied (virtual) probabilistic guards.

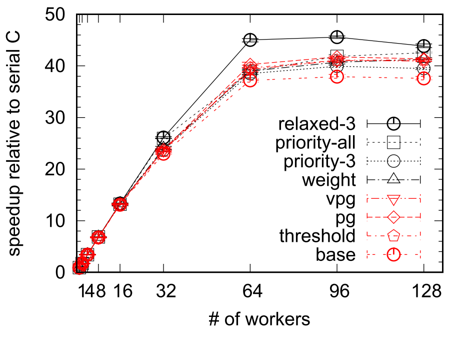
Figure 4 shows speedups (relative to serial C) of the force calculation. The first and third quartiles are shown as error bars using 22 runs (with distinct random seeds). The vertical axis, **speedup relative to serial C**, is calculated with t_S/t_P, where t_S is the execution time of the serial C program and t_P ranges over execution times of the Tascell program in parallel with P workers.

- **base** shows speedups (relative to serial C) of the force calculation with uniformly random selection.
- **pg** shows speedups of the force calculation with probabilistic guards; $p = min(1, kP\frac{m_{\mathrm{sub}}}{m_{\mathrm{all}}})$, 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 speedups of the force calculation with probabilistic guards based on thresholds; $p = 1$ if $kP\frac{m_{\mathrm{sub}}}{m_{\mathrm{all}}} > 1$, where k is 10; otherwise, $p = 0$.
- **vpg** uses virtual probabilistic guards; p is declared in the same manner as **pg**.
- **priority-all** shows speedups of the force calculation with priority-based selection ($\kappa = P$). Priority $w = m_{\mathrm{sub}}$.
- **priority-3** shows speedups of the force calculation with priority-based selection ($\kappa = 3$). Priority $w = m_{\mathrm{sub}}$.
- **weight** shows speedups of the force calculation with weight-based selection. Weight $w = m_{\mathrm{sub}}$.
- **relaxed-3** is based on **priority-all** but alleviates excessive steal backs by setting $\tau = 3$.

With a small number of workers, all strategies show near-ideal speedups. With 96 or 128 workers, most strategies have

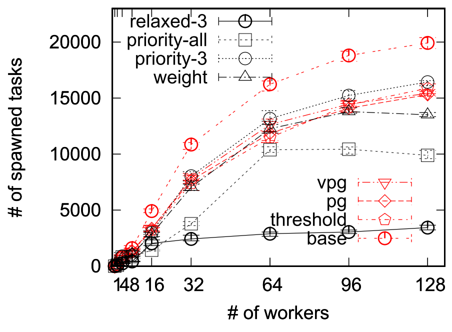


(a) 500,000 bodies

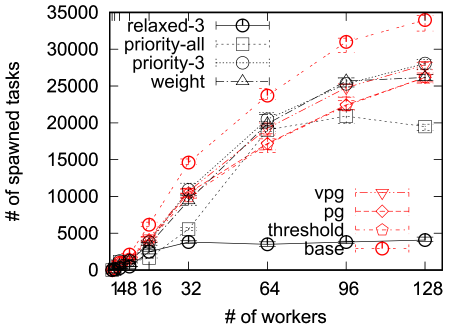


(b) 1,000,000 bodies

Fig. 4 Speedup of force calculation (relative to serial C).



(a) 500,000 bodies



(b) 1,000,000 bodies

Fig. 5 The number of spawned tasks in force calculation.

peaks. In Fig. 4 (a) with 96 (128) workers, **priority-all**, **priority-3**, **weight**, and **relaxed-3** show 20.3% (24.9%), 8.0% (9.0%), 13.8% (20.7%), and 24.5% (25.0%) improvements over **base** and 5.0% (7.4%), −5.6% (−6.2%), and −0.5% (3.8%), and 8.7% (7.5%) improvements over **pg**. In Fig. 4 (b) with 96 (128) workers, **priority-all**, **priority-3**, **weight**, and **relaxed-3** show 10.0% (12.4%), 4.5% (4.4%), 6.4% (9.5%), and 19.4% (16.5%) improvements over **base** and 0.7% (2.8%), −4.2% (−4.5%), −2.5% (0.1%), and 9.3% (6.5%) improvements over **pg**. When the retry-loop option is enabled, unlike [7], [8], [9], **pg** shows good speedups which are better than **vpg**.

Figure 5 shows the number of spawned tasks in the force calculation. We can confirm that **priority-all** (with **relaxed-3**) has improved the performance because the number of spawned tasks has decreased overall (significantly). In this setting, **vpg** and **priority-3** perform retry loops without individual probabilistic guards and spawn more tasks than **pg**. For **weight**, the number of spawned tasks is not as small as that of **priority-all**.

6.2 Evaluations in a Distributed Memory Environment

In the distributed memory environment, up to four processes run on co-processors (one process on each). We created 114 worker threads in each process. A *Tascell server* on host processors relays inter-node messages such as task requests, rejects, tasks, and results among four co-processors (nodes), where intra-node work stealing always precedes inter-node work stealing. Note that virtual probabilistic guards, priority-based selection, and weight-based selection can be used only for worker selection at each computing node: the Tascell server that receives a task request randomly selects a computing node to which the server forwards the request, as described in Section 2.1.

We employed the following benchmark programs:

- **Fib(*n*)** recursively computes the n-th Fibonacci number. Probability $p = min(1, \frac{n}{20})$, where n is the problem size of the current task. Priority or weight $w = \frac{n}{20}$.
- **Nq(*n*)** finds all solutions to the n-queens problem on the basis of backtrack search. Probability $p = \frac{n-(j+1)}{n-2}$ if $j > 1$; otherwise, $p = 1.0$, where j is the number of queens placed, and n is the number of all queens. Priority or weight $w = n - (j + 1)$.
- **Pen(*n*)** finds all solutions to the Pentomino problem with n pieces (using additional pieces and an expanded board for $n > 12$) on the basis of backtrack search. Probability $p = \frac{n-j}{n-2}$ if $j > 2$; otherwise, $p = 1.0$, where j is the number of pieces placed. Priority or weight $w = n - j$.
- **Cmp(*n*)** compares array elements a_i and b_j for all $0 \leq i, j < n$ using a cache-oblivious recursive algorithm. Probability $p = min(1, \frac{n}{500})$, where n is the problem size of the current task. Priority or weight $w = \frac{n}{500}$.
- **Histogram(*n*, N_R, *d*)** considers a multiset of n^d GCDs of all d-tuples each element of which is 2 or more and less than $2 + n$: $\{\gcd(i_1, i_2, \ldots, i_d) \mid 2 \leq i_1, i_2, \ldots, i_d < 2 + n\}$, and counts the number of occurrences of each GCD value (if the value is 2 or more) as a histogram of N_R ranks efficiently with pruning. Probability $p = \frac{n_{\max} d+n}{n_{\max} d_{\max}}$. Priority or weight $w = n_{\max} d + n$.

Of course, in Fib(n) and Cmp(n), we can simply use priority or weight $w = n$. We employed Fib(n) and Nq(n) as standard benchmark programs, Pen(n) as a significantly irregular program, and Cmp(n) and Histogram(n, N_R, d) as programs with large workspaces. We employ small problem sizes such as Fib(50), Nq(17), Pen(15), Cmp(150000), and Histogram(50, 50, 7) in order to clearly see speedup differences with a large number of workers.

Figures 6, **7**, **8**, **9** and **10** show speedups (relative to serial C) of benchmark programs.

- **priority-all+pval** is based on **priority-all** but sends the current maximum priority value within a node with task and task request messages to the Tascell server to make the server perform node-level priority-based selection over all nodes.

Priority-all shows considerable performance improvements; with 4 × 114 workers, **priority-all** is 4.2% better than **base** in Fib(50), 7.9% in Nq(17), 14.0% in Cmp(150000), and 5.6% in Histogram(50, 50, 7).

However, in Pen(15), **priority-all** is −1.2% better than **base**; that is, **priority-all** degrades the performance. With a very irregular search tree, a large subtree at some deep tree node may be found early by using randomness instead of an imprecise estimation.

Priority-all+pval and **priority-3** show some performance improvements, but they are less significant than **priority-all**. For **priority-all+pval**, node-level priority-based selection can only see somehow old values.

6.3 Evaluations of Alleviation of Excessive Work Stealing

Figures 11, **12**, **13**, **14** and **15** show measurement results when we alleviated excessive intra-node work stealing and excessive steal backs. The labels in these figures indicate execution settings

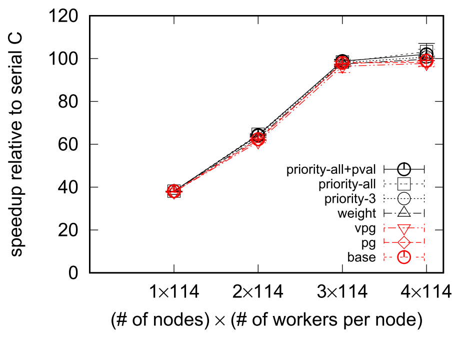


Fig. 6 Speedups of Fib(50) (relative to serial C).

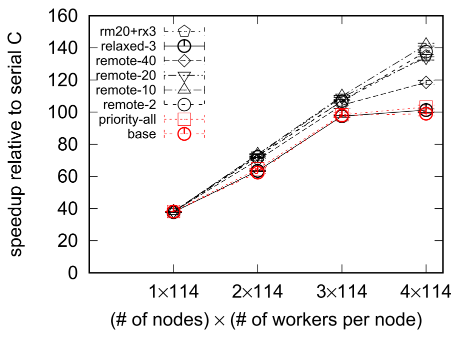


Fig. 11 Speedups of Fib(50) (relative to serial C).

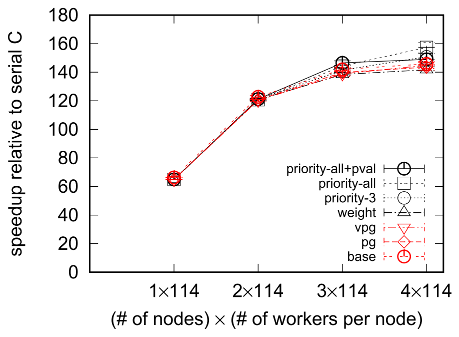


Fig. 7 Speedups of Nq(17) (relative to serial C).

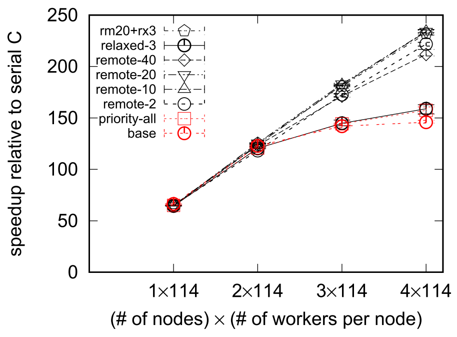


Fig. 12 Speedups of Nq(17) (relative to serial C).

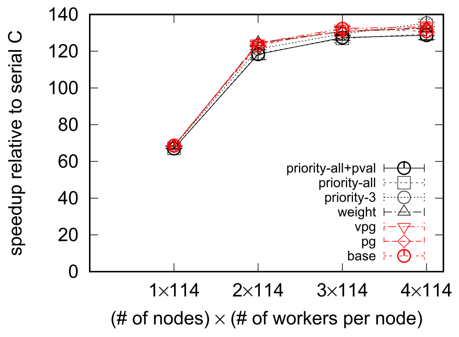


Fig. 8 Speedups of Pen(15) (relative to serial C).

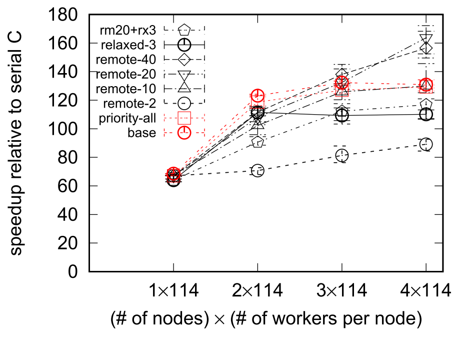


Fig. 13 Speedups of Pen(15) (relative to serial C).

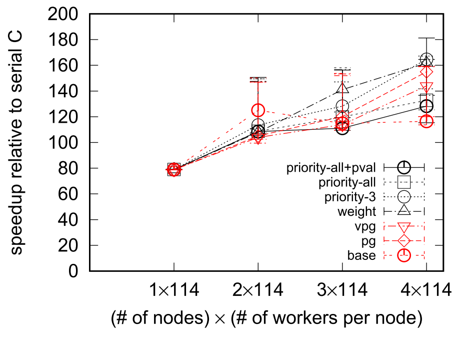


Fig. 9 Speedups of Cmp(150000) (relative to serial C).

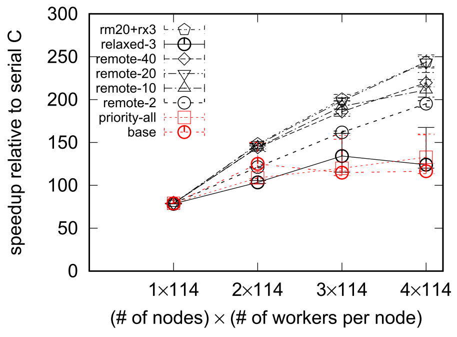


Fig. 14 Speedups of Cmp(150000) (relative to serial C).

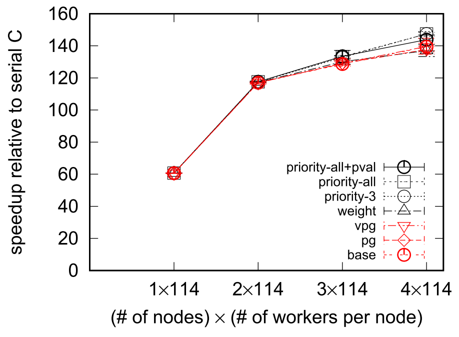


Fig. 10 Speedups of Histogram(50, 50, 7) (relative to serial C).

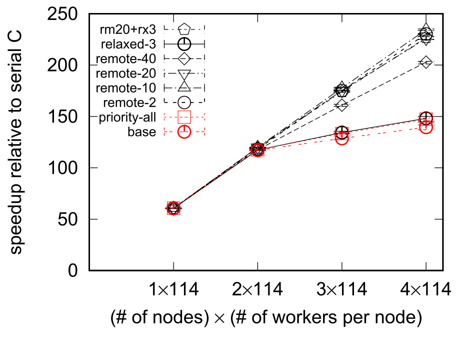


Fig. 15 Speedups of Histogram(50, 50, 7) (relative to serial C).

as follows.

- **remote-2**, **remote-10**, **remote-20**, and **remote-40** show speedups when we alleviated excessive intra-node work stealing by setting E to 2, 10, 20, and 40, respectively.
- **relaxed-3** shows speedups when we alleviated excessive steal backs by setting $\tau = 3$. Alleviation of excessive intra-node work stealing is not employed ($E = \infty$).
- **rm20+rx3** shows speedups when we alleviated both excessive intra-node work stealing and excessive steal backs by setting $E = 20$ and $\tau = 3$.

In all of these settings, priority-based selection ($\kappa = 114$) was used for worker selection inside each computing node.

Remote-20 shows considerable performance improvements; with 4×114 workers, **remote-20** is 29.7% better than **priority-all** in Fib(50), 48.8% in Nq(17), 26.5% in Pen(15), 83.4% in Cmp(150000), and 53.2% in Histogram(50, 50, 7).

Relaxed-3 shows negligible/positive/negative performance improvements; with 4×114 workers, **relaxed-3** is −1.8% better than **priority-all** in Fib(50), 1.0% in Nq(17), −14.8% in Pen(15), −6.6% in Cmp(150000), and 0.4% in Histogram(50, 50, 7). **Rm20+rx3** shows negligible/negative performance improvements over **remote-20**; with 4×114 workers, **rm20+rx3** is −28.6% better than **remote-20** in Pen(15). In these evaluations, **remote-20** alone suffices.

7. Related Work

Work-stealing frameworks are typically implemented as multithreaded languages [1], [2], [3], [4], [5] or libraries [19]. Unlike multithreaded languages, Tascell [6] provides logical-thread-free on-demand concurrency; a worker performs a computation sequentially unless requested, and a new task is spawned only when requested.

Duran et al. proposed an adaptive cut-off for task parallelism that determines which tasks should be pruned [20]. Wang et al. proposed adaptive task creation [21]. These techniques enable adaptive creation time cut-offs. In Tascell, the granularity control can be applied at steal time, and a thief can use some global information over victims when using virtual probabilistic guards [7], [8] or the proposed priority- and weight-based selection strategies.

Shiina and Taura proposed Almost Deterministic Work Stealing (ADWS) [22], which primarily addresses the issue of data locality. ADWS would reduce the number of steals as well. Yasugi et al. proposed a Parallel Execution model based on Hierarchical Omission (HOPE) [17], which primarily addresses fault tolerance; every HOPE worker starts a redundant (idempotent) computation sequentially with its own predetermined *order* and dynamic work omission.

Various types of priority-based selection are commonly used for victim selection. For example, OpenJDK [23] employs a strategy similar to **priority**-2 for parallel GC (Garbage Collection); Horie et al. proposed a strategy similar to **priority**-κ with automatic adjustment of κ based on the number of GC workers [24]. Unlike work stealing for parallel GC, our proposed Tascell framework allows the users (application programmers) to let each worker estimate and declare, as a real number, the amount of remaining work so that declared values are used as "priorities" or "weights".

Okuno et al. proposed work stealing strategies for a certain program that extracts connected subgraphs with common itemsets in distributed memory environments [25]. In their work stealing strategies, to increase the frequency of task requests to outside nodes, a thief sends a task request to external nodes if the number of uncompleted tasks taken from external nodes (except tasks taken by stealing back) is less than a threshold. In addition, to prevent workers from stealing back a small task, a worker waits for the result for a certain period of time before sending a stealing back request to an external node.

Feeley proposed victim selection from all candidates instead of "steal backs" [26]. To prevent stack overflow, workers with stacks of the maximum size must wait for the replies by busy-waiting.

8. Conclusion

This paper proposed four steal strategies for work-stealing frameworks, namely, two new types of priority- and weight-based strategies, a non-hierarchical strategy to alleviate excessive intra-node work stealing, and a relaxed strategy to alleviate excessive "steal backs", and implemented them for a parallel language, Tascell. To reduce the total task-division cost, each worker can estimate the work amount of its current task as a real number to be used as a priority or weight upon victim selection.

From the performance evaluation of the Barnes–Hut algorithm, we confirmed that the proposed priority- and weight-based strategies showed better performance than uniformly random selection. Furthermore, priority-based strategy (and a relaxed strategy) also showed better performance than (virtual) probabilistic guards that we previously proposed. We conducted a performance evaluation in a distributed memory environment with five Tascell programs and mostly confirmed the effectiveness of the proposed strategies.

Comparing the priority- and weight-based steal strategies, we could not clarify cases where the weight-based strategy would be better. At least, we can say that the priority-based strategy among all workers is worth trying in most cases.

For future work, we would like to enhance our implementation to support more combinations of the proposed and existing strategies. First, as a combination of priority- and weight-based strategies, the future implementation may perform priority-based selection of top m ($m > 1$) candidates from all workers (by using some data structures) and then perform weight-based selection from the top m candidates. This may improve performance if over-concentration is problematic. Second, probabilistic guards with a retry loop can be combined with priority- and/or weight-based strategies.

The proposed framework provides Tascell programmers with various opportunities to improve performance. For future work, we (as representative users) would like to explore the best use of the opportunities to figure out the best performance users can achieve, for example, by adjusting the numbers of candidates in priority-based selection; such investigations would be useful for developing automatic adjustment.

We will apply the proposed strategies to other frameworks, such as Cilk, for future work.

Acknowledgments We thank Ken Matsui for the development of the parallel implementation of Treecode 1.4 (a fast "highly serial" version of the Barnes–Hut algorithm) in Tascell. This work was supported in part by JSPS KAKENHI Grant Numbers JP19H04087 and JP17K00099.

References

[1] Mohr, E., Kranz, D.A. and Halstead, Jr., R.H.: Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs, *IEEE Trans. Parallel and Distributed Systems*, Vol.2, No.3, pp.264–280 (1991).

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

[3] Feeley, M.: A Message Passing Implementation of Lazy Task Creation, *Proc. Intl. Workshop on Parallel Symbolic Computing: Languages, Systems, and Applications*, Lecture Notes in Computer Science, No.748, pp.94–107, Springer-Verlag (1993).

[4] Taura, K., Tabata, K. and Yonezawa, A.: StackThreads/MP: Integrating Futures into Calling Standards, *Proc. ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming* (*PPoPP '99*), pp.60–71 (1999).

[5] IBM Research: X10: Performance and Productivity at Scale, available from ⟨http://x10-lang.org/⟩.

[6] Hiraishi, T., Yasugi, M., Umatani, S. and Yuasa, T.: Backtracking-based Load Balancing, *Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (*PPoPP 2009*), pp.55–64 (2009).

[7] Yoritaka, H., Matsui, K., Yasugi, M., Hiraishi, T. and Umatani, S.: Extending a Work-Stealing Framework with Probabilistic Guards, *Proc. 45th International Conference on Parallel Processing Workshops* (*ICPPW 2016*) (*9th International Workshop on Parallel Programming Models and Systems Software for High-End Computing P2S2 2016, held in conjunction with ICPP 2016*), pp.171–180 (2016).

[8] Yoritaka, H., Matsui, K., Yasugi, M., Hiraishi, T. and Umatani, S.: Probabilistic guards: A mechanism for increasing the granularity of work-stealing programs, *Parallel Computing*, Vol.82, pp.19–36 (online), DOI: 10.1016/j.parco.2018.06.003 (2019).

[9] Nakashima, R., Yoritaka, H., Yasugi, M., Hiraishi, T. and Umatani, S.: Extending a Work-Stealing Framework with Priorities and Weights, *Proc. Workshop on Irregular Applications: Architectures and Algorithms* (*IA3 2019*) (*held in conjunction with SC 2019*), pp.9–16 (online), DOI: 10.1109/IA349570.2019.00008 (2019).

[10] Wagner, D.B. and Calder, B.G.: Leapfrogging: A Portable Technique for Implementing Efficient Futures, *Proc. Principles and Practice of Parallel Programming* (*PPoPP'93*), pp.208–217 (1993).

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

[12] Barnes, J.E.: Treecode Guide, available from ⟨http://www.ifa.hawaii.edu/~barnes/treecode/treeguide.html⟩.

[13] Barnes, J.E.: A Modified Tree Code: Don't Laugh; It Runs, *Journal of Computational Physics*, Vol.87, pp.161–170 (1990).

[14] Yasugi, M., Hiraishi, T., Shinohara, T. and Yuasa, T.: L-Closure: A Language Mechanism for Implementing Efficient and Safe Programming Languages, *IPSJ Trans. Programming*, Vol.49, No.SIG 1 (PRO 35), pp.63–83 (2008). (in Japanese).

[15] Yasugi, M., Hiraishi, T. and Yuasa, T.: Lightweight Lexical Closures for Legitimate Execution Stack Access, *Proc. 15th International Conference on Compiler Construction* (*CC 2006*), Lecture Notes in Computer Science, No.3923, pp.170–184, Springer-Verlag (2006).

[16] Yasugi, M., Ikeuchi, R., Hiraishi, T. and Komiya, T.: Evaluating Portable Mechanisms for Legitimate Execution Stack Access with a Scheme Interpreter in an Extended SC Language, *Journal of Information Processing*, Vol.27, pp.177–189 (online), DOI: 10.2197/ipsjjip.27.177 (2019).

[17] Yasugi, M., Muraoka, D., Hiraishi, T., Umatani, S. and Emoto, K.: HOPE: A Parallel Execution Model Based on Hierarchical Omission, *Proc. 48th International Conference on Parallel Processing* (*ICPP 2019*), pp.77:1–77:11 (online), DOI: 10.1145/3337821.3337899 (2019).

[18] Breuel, T.: Lexical Closures for C++, *Usenix Proceedings, C++ Conference* (1998).

[19] Intel Corporation: *Intel Threading Building Block Reference Manual* (2007), available from ⟨http://threadingbuildingblocks.org/⟩.

[20] Duran, A., Corbalán, J. and Ayguadé, E.: An Adaptive Cut-off for Task Parallelism, *Proc. 2008 ACM/IEEE Conference on Supercomputing, SC '08*, pp.36:1–36:11 (2008).

[21] Wang, L., Cui, H., Duan, Y., Lu, F., Feng, X. and Yew, P.-C.: An Adaptive Task Creation Strategy for Work-stealing Scheduling, *Proc. 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '10*, pp.266–277 (online), DOI: 10.1145/1772954.1772992 (2010).

[22] Shiina, S. and Taura, K.: Almost Deterministic Work Stealing, *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis, SC '19*, pp.1–16 (online), DOI: 10.1145/3295500.3356161 (2019).

[23] Oracle Corporation and/or its affiliates: OpenJDK, available from ⟨https://openjdk.java.net/⟩.

[24] Horie, M., Ogata, K., Takeuchi, M. and Horii, H.: Scaling up Parallel GC Work-Stealing in Many-Core Environments, *Proc. International Symposium on Memory Management, ISMM 2019*, pp.27–40 (online), DOI: 10.1145/3315573.3329985 (2019).

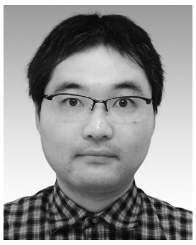
[25] Okuno, S., Hiraishi, T., Nakashima, H., Yasugi, M. and Sese, J.: Parallelization of Extracting Connected Subgraphs with Common Itemsets in Distributed Memory Environments, *Journal of Information Processing*, Vol.25, pp.256–267 (online), DOI: 10.2197/ipsjjip.25.256 (2017).

[26] Feeley, M.: Lazy Remote Procedure Call and its Implementation in a Parallel Variant of C, *Proc. International Workshop on Parallel Symbolic Languages and Systems*, Lecture Notes in Computer Science, No.1068, pp.3–21, Springer-Verlag (1995).

Ryusuke Nakashima was born in 1996. He received his Bachelor's degree in Computer Science and Systems Engineering (Artificial Intelligence) from Kyushu Institute of Technology in 2019. Since 2019, he is a master course student at Kyushu Institute of Technology. His research interests include programming languages and parallel processing.

Masahiro Yasugi was born in 1967. He received his B.E. in Electronic Engineering, his M.E. in Electrical Engineering, and his Ph.D. in Information Science from the University of Tokyo in 1989, 1991, and 1994, respectively. In 1993–1995, he was a fellow of the JSPS (at the University of Tokyo and the University of Manchester). In 1995–1998, he was a research associate at Kobe University. In 1998–2012, he was an associate professor at Kyoto University. Since 2012, he is a professor at Kyushu Institute of Technology. In 1998–2001, he was a researcher at PRESTO, JST. His research interests include programming languages and parallel processing. He is a member of IPSJ, ACM, and the Japan Society for Software Science and Technology. He was awarded the 2009 IPSJ Transactions on Programming Outstanding Paper Award.

Hiroshi Yoritaka was born in 1991. He received his Bachelor and Master of Computer Science and Systems Engineering from Kyushu Institute of Technology in 2015 and 2017 respectively. Since 2017, he works for Ad-Sol Nisshin Corporation. His research interests include parallel processing.



Tasuku Hiraishi was born in 1981. He received his B.E. in Information Science in 2003, Master of informatics in 2005, and a Ph.D. in informatics in 2008, all from Kyoto University. In 2007–2008, he was a JSPS fellow at Kyoto University. Since 2008, he has been working at Kyoto University as an assistant professor at Supercomputing Research Laboratory, Academic Center for Computing and Media Studies, Kyoto University. His research interests include parallel programming languages and high performance computing. He won the IPSJ Best Paper Award in 2010. He is a member of IPSJ, JSSST, and ACM.

Seiji Umatani was born in 1974, and received a B.E. degree in information science, and M.E. and Ph.D. degrees in informatics from Kyoto University, Kyoto, Japan, in 1999, 2001, and 2004, respectively. In 2004–2005, he was a research staff member in the Graduate School of Informatics at Kyoto University, and he was an assistant professor in 2005–2019. He is currently an associate professor at Kanagawa University. His current research interests include programming languages, compilers, DSL, and program analysis. He is a member of ACM and the Japan Society for Software Science and Technology.