Backtracking-based Load Balancing

Tasuku Hiraishi

Academic Center for Computing and Media Studies,
Kyoto University
{tasuku}@media.kyoto-u.ac.jp

Masahiro Yasugi Seiji Umatani
Taiichi Yuasa

Graduate School of Informatics, Kyoto University
{yasugi, umatani, yuasa}@kuis.kyoto-u.ac.jp

Abstract

High-productivity languages for parallel computing become more important as parallel environments including multicores become more common. Cilk is such a language. It provides good 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. This paper proposes a "logical thread"-free framework called *Tascell*, which achieves a higher performance and supports a wider range of parallel environments including clusters without loss of productivity. A Tascell worker spawns a "real" task only when requested by another idle worker. The worker performs the spawning by temporarily "backtracking" and restoring its oldest task-spawnable state. Our approach eliminates the cost of spawning/managing logical threads. It also promotes the reuse of workspaces and improves the locality of reference since it does not need to prepare a workspace for each concurrently runnable logical thread. Furthermore, Tascell enables elegant and highly-efficient backtrack search algorithms with delayed workspace copying. For instance, our 16-queens problem solver is 1.86 times faster than Cilk on a system with two dual-core processors. Our approach also enables a single program to run in both shared and distributed memory environments with reasonable efficiency and scalability.

Categories and Subject Descriptors D.3.3 [*PROGRAMMING LANGUAGES*]: Language Constructs and Features—Concurrent programming structures

General Terms Design, Languages, Performance

Keywords load balancing, parallel computing, backtracking, backtrack search

1. Introduction

For efficient parallel computing, all computing resources—such as cores in cluster nodes—in a system are expected to have their own work at any given time. However, it is difficult to predict appropriate work allocation statically in heterogeneous or dynamically varying environments involving multitasking operating systems and/or dynamically joining/leaving computing resources. Irregular applications, such as tree-recursive algorithms and backtrack search algorithms, also make the prediction difficult. In such cases, dynamic load balancing, where a task (a piece of work) is dynamically allocated to idle computing resources, is effective. Note that the entire computation should be divided into larger tasks in order to reduce the total division/allocation costs.

For shared memory environments, Cilk [5] provides good load balancing in addition to award-winning overall productivity [11]. It succeeds in keeping all workers busy by creating plenty of "logical" threads and adopting the oldest-first work stealing strategy. Creation of logical threads and their synchronization can simply be specified with the keywords `spawn` and `sync` as extensions to C. *Workers* are *OS threads* provided as virtual computing resources. Usually, the number of workers does not exceed the number of underlying computing resources so that workers actually run in parallel. Cilk employs the implementation technique called LTC (Lazy Task Creation) [12], in which each worker spawns plenty of logical threads and schedules them internally and thus efficiently. An idle worker (thief) may steal a logical thread from another worker (victim). That is, logical threads are used as tasks dynamically allocated to idle workers. When a logical thread recursively spawns offspring logical threads, the adoption of the oldest-first work stealing strategy is effective in making tasks larger.

This paper proposes a "logical thread"-free framework called *Tascell*, which achieves a higher performance and supports a wider range of parallel environments including clusters without loss of productivity. A Tascell worker spawns a "real" task only when requested by another idle worker. The worker performs the spawning by temporarily "backtracking" and restoring its oldest task-spawnable state.

The contributions of this paper are twofold:

- We propose a new idea for dynamic load balancing, which is based on "backtracking" and does not use "logical" threads.

  A worker performs a computation sequentially with an ability to perform backtracking-based temporary restoration of task-spawnable states.

- We discuss how to implement this new idea as an efficient dynamic load balancing programing/execution framework. We also discuss its effectiveness in terms of suitable applications and environments.

  Our approach eliminates the cost of spawning/managing logical threads. It also promotes the reuse of workspaces and improves the locality of reference since it does not need prepare a workspace for each concurrently runnable logical thread.

  Our framework allows programmers to write undo-redo operations to be executed in backtracking. This enables elegant and highly-efficient backtrack search algorithms with delayed workspace copying. For instance, our 16-queens problem solver is 1.86 times faster than Cilk on a system with two dual-core processors.

PPoPP'09, February 14–18, 2009, Raleigh, North Carolina, USA.

```
int fib (int n) {
 if (n <= 2) return 1;
 {
   int s1, s2;
   s1 = fib(n - 1);
   s2 = fib(n - 2);
   return s1 + s2;
 }
}
```

Figure 1. C program for Fibonacci.

```
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;
}
```

Figure 2. C program for Pentomino.

In addition, we adopted message passing for communication among workers that exchange tasks and results as serialized task objects. This allows a single program to run in both shared and distributed (and also hybrid) memory environments with reasonable efficiency and scalability.

2. Motivation

We introduce two tree-recursive algorithms as examples. These algorithms are traditionally difficult to parallelize with low overhead, and our approach is effective (even) for them. Note that the second example supports parallel loops in a tree-recursive manner. By using these examples, we discuss the difficulty in this section and explain the details of our proposal in the next section.

The first example is a doubly recursive algorithm for computing the n-th term of the Fibonacci sequence. Figure 1 shows a sequential C program for it. In each function call, computations of fib$(n-1)$ and fib$(n-2)$ can be executed in parallel. This algorithm has no practical meaning, but this example is often used (e.g., in [3, 4, 5, 6, 9, 12, 18, 19]) for evaluating parallel languages; since each function call has little actual work, the measured overhead well represents the worst case overhead for similar tree-recursive algorithms.

The second example is a search algorithm for finding all possible solutions to the Pentomino puzzle. A pentomino consists of five squares attached edge-to-edge. There are twelve pentominos of different shapes. The Pentomino puzzle involves filling the 6×10 rectangular board with the twelve pentominos. This problem represents many similar search problems. Figure 2 shows a sequential C program for this problem. Each function call iterates through unused pieces (the outermost loop) and their directions (the inner loop). Parallelization seems applicable to the outermost loop. However, there is an important difference from the Fibonacci program; this program does backtrack search where states of the board and the pieces are stored in workspaces: a piece is set at the next available position by one-step extension and removed by backtracking.

```
// The structure of task objects.
struct tfib {
  int n;  // input
  int r;  // output
};
// The entry point of a task.
void exec_fib_task (struct tfib *pthis)
{ pthis->r = fib (pthis->n); }

int fib (int n) {
 if (n <= 2) return 1;
 {
   int s1, s2;
   if (choose not to spawn?) {
     s1 = fib(n - 1);
     s2 = fib(n - 2);
   } else {
     Allocate a workspace of struct tfib as this.
     this.n = n - 2;  // put the input value
     Send this as a newly spawned task.
     s1 = fib(n - 1);
     Wait and receive the result of this.
     s2 = this.r;      // get the output value
     Deallocate this workspace.
   }
   return s1 + s2;
 }
}
```

Figure 3. Straightforward task-parallel program for Fibonacci.

One might simply think that one can achieve efficient load balancing by restricting task creation according to the "latest" state (e.g., the number of spawned tasks). Such a straightforward task-parallel program for Fibonacci can be written as in Figure 3. In fib(n), each worker *chooses* whether it executes `fib(n-2)` by itself or it spawns a `fib(n-2)` task. For efficient load balancing, each task should be as large as possible so that a *minimum sufficient* number of tasks are created to keep all workers busy during the entire running time. That is, for each task, its worker should choose to spawn a *proper* number of tasks in the early stage and then choose not to spawn any more tasks except for adjusting the completion time. Such a strategy is infeasible without precise information (prediction) about the entire execution (not the "latest" information at the choice point). Thus, this straightforward approach does not work.

Figure 4 shows a straightforward task-parallel program for Pentomino. For parallelization, the outer `for` loop in Figure 2 is replaced with a `PAR_LOOP` macro. In `PAR_LOOP`, each worker *chooses* whether it performs all iterations or it spawns a task for the upper half of iterations. (In the latter case, it has another choice on the remaining lower half of iterations with a recursive application of the `PAR_LOOP` macro.) This straightforward approach does not work since it requires an infeasible strategy as in the case of the Fibonacci program.

In the following sections, we propose our approach with a feasible strategy for efficient load balancing. Note that the worker that executes a spawned task often requires its own initialized (copied) workspace as in Figure 4. This motivated us to make an additional innovation in our approach.

3. Our approach

We propose a programming and execution framework called "Tascell." Tascell stands for *task cell*, which indicates that running tasks are divided like biological cells. In Tascell, we can spawn a task *lazily* by using *backtracking*.

```
// The structure of task objects.
// Each worker has to have its own board for parallelization.
struct pentomino {
  int s;      // output
  int k, i0, i1, i2;
  int a[12]; // manage unused pieces
  int b[70]; // the board, with (6+sentinel) × 10 cells
};
exec_pentomino_task (struct pentomino *pthis)
{ pthis->s = search(pthis->k, pthis->i0, pthis->i1,
                    pthis->i2, pthis); }

#define PAR_LOOP (_i1, _i2, _body) {
   if (choose not to spawn?) {
     for(; _i1 < _i2; _i1++) _body
   } else {
     int _ih = (_i1 + _i2) / 2;
     int i1 = _ih;  // range for the new sub-task i1--i2
     int i2 = _i2;
     Allocate a workspace of struct pentomino as this.
     { // put task inputs for upper half iterations
       copy_piece_info (this.a, tsk->a); // copy the
       copy_board (this.b, tsk->b);      // workspace
       this.k = k; this.i0 = j0;
       this.i1 = i1; this.i2 = i2;
     }
     Send this as a newly spawned task.
     // lower half iterations (expanded n times for n-bit int)
     PAR_LOOP (_i1, _ih, _body)
     Wait and receive the result of this.
     s += this.s;    // get the result
     Deallocate this workspace.
   }
}
// Try from the j1-th piece to the j2-th 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 j1, int j2,
            struct pentomino *tsk)
{
  int s=0; // the number of solutions
  int p=j1;
  PAR_LOOP(p, 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;
      Set the ap-th piece onto the board tsk->b and update tsk->a.
      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);
      Backtrack, i.e., remove the ap-th piece from tsk->b
      and restore tsk->a.
    }
  })
  return s;
}
```

Figure 4. Straightforward task-parallel program for Pentomino.

The sequential computation of the C program in Figure 1 (Figure 2) is outlined as a depth-first, left-to-right traversal of the *invocation tree*. Notice that the straightforward task-parallel program in Figure 3 (Figure 4) involves the same traversal *if* its worker always chooses not to spawn a task.

In Tascell, the worker always chooses not to spawn *at first*, but when it receives a task request, it spawns a task as if *it changed the past choice*. That is, as is shown in Figure 5 (Figure 6),

1. it *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 (restores the time), and

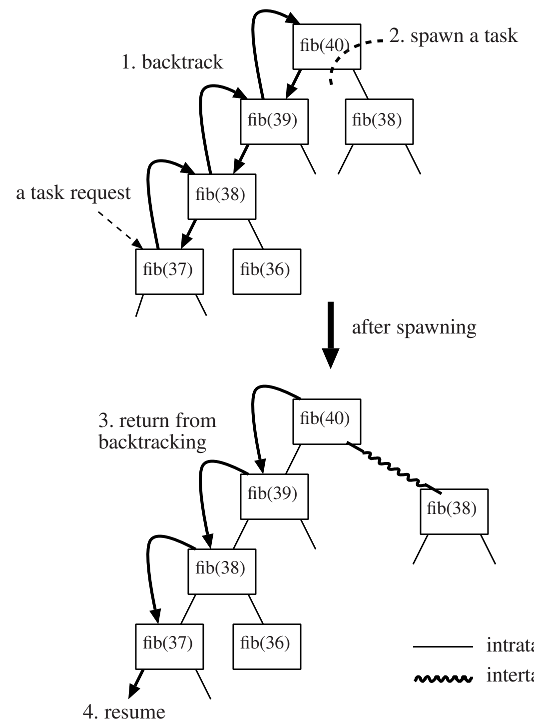

Figure 5. Spawning a task lazily while computing fib(40). When a Tascell worker detects a task request (at fib(37)), it (1) backtracks to the oldest task-spawnable point, (2) spawns a task for fib(38), (3) returns from backtracking, and (4) resumes its own computation.

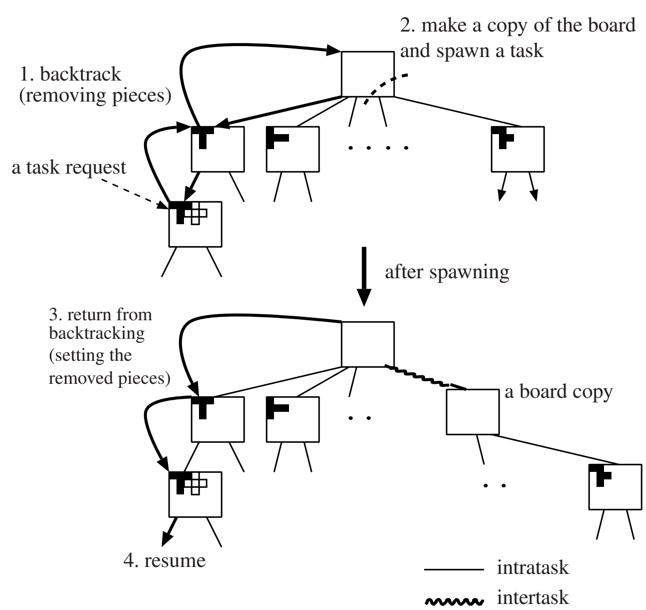

Figure 6. Spawning a task lazily while performing backtrack search for Pentomino. Unlike in Figure 5, (1) the backtracking step includes undo operations (i.e., removing pieces). (2) The spawning-half-iterations step includes making a copy of the temporarily restored board. (3) The returning-from-backtracking step includes redo operations (i.e., setting pieces).

4. then it resumes its own task.

Notice that we can spawn a larger task (as is a fib(38) subtree in the lower right part of Figure 5), in general, by backtracking to the oldest task-spawnable choice point.[1]

A Pentomino worker performs a sequential computation efficiently with its own workspace by setting a piece and by removing the piece (i.e., backtracking or *undo*ing) across search steps. When the worker spawns a task, it must copy (part of) the "current" contents of its workspace into a newly allocated space for the new task as in Figure 4. In our approach, the "current" contents should be equal to the past contents at the time of the past choice. As is shown in Figure 6, the worker can recover the past contents by performing proper *undo* operations along with backtracking as part of Step 1, it spawns a task with a copy of its workspace at Step 2, and then it performs proper *redo* operations as part of Step 3 in order to resume its own task at Step 4.

To address the problem of *load-based inlining*, which is essentially the straightforward approach in Section 2, fine-grain multithreaded languages such as Cilk [5] and MultiLisp [7] also use a technique called Lazy Task Creation (LTC) [12]. Our approach differs from LTC in the following manner:

- Our worker performs a sequential computation unless it receives a task request. Because no *logical threads* are created as potential tasks, the cost of managing a queue for them can be eliminated.
- In multithreaded languages, each (logical) thread requires its own workspace. In contrast, our worker can reuse a single workspace while it performs a sequential computation to improve the locality of reference and achieve a higher performance.
- When we implement a backtrack search algorithm in multithreaded languages, each thread often needs each its own copy of its parent thread's workspace. In contrast, our worker can delay copying between workspaces by using backtracking.
- Our approach supports (heterogeneous) distributed memory environments (including mixed-endian environments) without using distributed shared memory systems.

Note that LTC assumes that the number of really created tasks (and steals) is incomparably smaller than the number of logical threads. Our approach also assumes that the number of really spawned tasks (and steals) is very small. This assumption justifies our approach, which accepts higher work-stealing (backtracking) overheads in order to achieve lower serial overheads than more conventional LTC such as Cilk.

We may use additional constructs in order to specify how to perform backtracking and undo-redo operations. These constructs are detailed in Section 4.2.

4. Tascell framework

We designed and implemented a framework to realize our idea. The Tascell framework consists of a Tascell server and a compiler for the Tascell language.

4.1 Overview

Figure 7 shows a multistage overview of the Tascell framework. Compiled Tascell programs are executed on one or more computation nodes. Each computation node has one or more worker(s) in the shared memory environment (the number can be specified as a runtime option).

[1] Tascell can be extended such that, beyond the statistical assumption, we may spawn larger tasks by individually considering all task-spawnable points using additional expected task information from users, analyzers, and/or profilers.

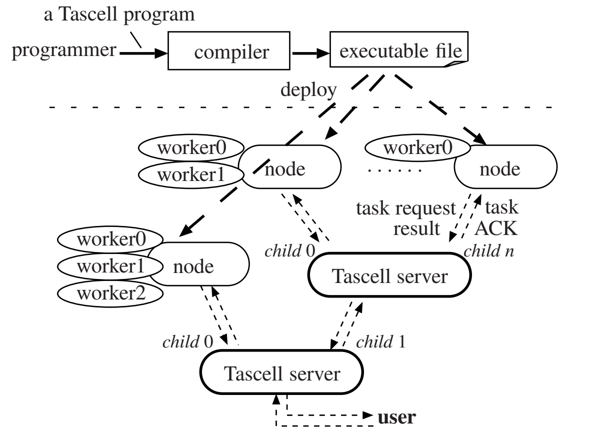

Figure 7. Multistage overview of the Tascell framework.

```
// The definition of a task named tfib
task tfib {
  in:  int n;  // input
  out: int r;  // output
};
// The entry point of tfib.
// The task object this is declared implicitly.
task_exec tfib
{ this.r = fib (this.n); }

worker int fib (int n) {
 if (n <= 2) return 1;
 {
   int s1, s2;
   do_two          // construct in Tascell
     s1 = fib(n - 1);
     s2 = fib(n - 2);
   tfib {  // The task object this is declared implicitly.
     // put part (performed before sending a task)
     { this.n = n - 2; }
     // get part (performed after receiving the result)
     { s2 = this.r; }
   }               // end of do_two
   return s1 + s2;
 }
}
```

Figure 8. Tascell program for Fibonacci.

For good load balancing, idle workers should request tasks of *loaded* workers. An idle worker sends a task request to either a specific worker or any worker. Intranode (Internode) messages are relayed by Tascell runtime systems (Tascell servers), which choose loaded workers (nodes) for "to any" task requests. Such a series of messages is exchanged automatically; programmers need not (and cannot) treat each message directly.

Each task or its result is transmitted as a task object whose structure is defined in a Tascell program. If a request is from the same node, (the pointer to) the object can be passed quickly via shared memory, otherwise the object is transmitted as a serialized message via Tascell servers.

4.2 Tascell Language

The Tascell language is an extended C language. Figures 8 and 9 are examples of Tascell programs. (Tascell extensions are underlined.)

Programmers can write a worker program with new constructs in Tascell, starting with an existing sequential program. Tascell

```
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.r = 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
   }
  }
 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;
}
```

Figure 9. Tascell Program for Pentomino.

has constructs for defining a task and for specifying potential task division with optional temporary undo/redo operations.

4.2.1 Task definition

A top-level task declaration:

`task` *task-name* `{` [`in:`|`out:`] *struct-declaration* ⋯ `};`

gives the structure of *task-name* task objects. For instance, "`task tfib {in: int n; out: int r};`" in Figure 8 declares the structure of `task tfib` objects. The syntax is the same as that for the definitions of `struct`s, except that we may specify an `in:` or `out:` attribute for each field. A Tascell compiler uses attributes to construct default send/receive methods of the task. In addition, we can add user-defined send/receive methods in order to skip transmitting (part of) inputs/outputs selectively or allocating/freeing a workspace (the details are omitted in this paper because of space limitation).

Definition of entry points A top level declaration

`task_exec` *task-name* `{` *body* `}`

defines the computation of a *task-name* task. In the *body* ("`this.r =fib(this.n);`" in Figure 8), we can refer to the task object by the keyword `this`, which includes an input of the task in some fields, and we should set the result of the computation into appropriate fields. In addition, we can call `worker` functions in the *body*.

4.2.2 Worker functions

In `worker` functions, which are specified with the keyword `worker` (like `cilk` procedures in Cilk), we can use Tascell's task division constructs as explained below. Only `worker` functions (and `task_exec` bodies) can call `worker` functions.

4.2.3 Constructs for task division

A statement:

`do_two` *statement*$_1$ *statement*$_2$
task-name `{` *statement*$_{put}$ *statement*$_{get}$ `}`

indicates that a computation in *statement*$_2$ ("`fib(n-2)`" in Figure 8) may be spawned during the execution of *statement*$_1$ ("`s1=fib(n-1);`" in Figure 8).[2] More precise steps are as follows:

1 The worker executes *statement*$_1$ with an implicit *task request handler*. If it invokes this handler with a task request after backtracking, it divides the current task by spawning a new *task-name* task, setting the fields of the new task object by *statement*$_{put}$ ("`this.n=n-2;`" in Figure 8), and then sending it to the task requester before returning from the backtracking. Here, a computation in *statement*$_2$ is packed as the task object.

2a If the task request handler for `do_two` is not invoked during the execution of *statement*$_1$, *statement*$_2$ is executed.

2b Otherwise, the worker skips *statement*$_2$, waits for the result of the spawned task, and then merges the result by executing *statement*$_{get}$ ("`s2=this.r;`" in Figure 8). In order not to be idle, the worker should request and execute other tasks while waiting for the result. In Tascell, the worker simply requests tasks of the task requester in Step 1 to save its stack size [22].

The identifier *task-name* specifies the type of a task to be created. The keyword `this` can be used in *statement*$_{put}$ and *statement*$_{get}$ to refer to the task object. We should initialize a task in *statement*$_{put}$ by assigning values to input fields and can obtain the result of the task in *statement*$_{get}$ by referring to output fields. This series of operations should be equivalent to *statement*$_2$.

For dividing an iterative computation, Tascell has a parallel `for` loop construct syntactically denoted by:

`for(int` *identifier* `:` *expression*$_{from}$ `,` *expression*$_{to}$`)` *statement*$_{body}$
task-name `(int` *identifier*$_{from}$ `, int` *identifier*$_{to}$`)`
`{` *statement*$_{put}$ *statement*$_{get}$`}`

[2] Although making work amounts of two statements as equal as possible is a good practice, imbalance and uncertainty are permitted to some degree. For example, in a simple situation with two workers, the total amount of unprocessed work decreases rapidly as a *product* of ε_i when the i-th division makes two workers have work amounts in the ratio of $1+\varepsilon_i : 1-\varepsilon_i$ $(0 \leq \varepsilon_i < 1)$.

For example, Figure 9 employs a parallel `for` statement of "`for (int p: j1, j2)` {...} `pentomino (int i1,int i2) {`{...}`{s+=this.s}}`." This iterates *statement*$_{body}$ over integers from *expression*$_{from}$ (inclusive) to *expression*$_{to}$ (exclusive). When the implicit task request handler (available during the iterative execution of *statement*$_{body}$) is invoked, the upper half of the remaining iterations are spawned as a new *task-name* task. The actual assigned range can be referred to in *statement*$_{put}$ by *identifier*$_{from}$ and *identifier*$_{to}$. The worker handles the result of the spawned task by executing *statement*$_{get}$.

Tascell has a `dynamic_wind` construct as in the Scheme language [10] for defining undo/redo operations, syntactically denoted by:

`dynamic_wind` *statement*$_{before}$ *statement*$_{body}$ *statement*$_{after}$.

The worker basically executes *statement*$_{before}$ ("set a piece" in Figure 9 as "do"), *statement*$_{body}$, and *statement*$_{after}$ ("remove the piece" in Figure 9 as "undo") in this order. However, during the execution of *statement*$_{body}$, *statement*$_{after}$ is also executed as an "undo" clause *before* an attempt to invoke an older task request handler. *Statement*$_{before}$ is also executed as a "redo" clause *after* the attempt.

Backtracking-based task division `Do_two`, parallel `for`, and `dynamic_wind` statements may be nested *dynamically* in their *statement*$_1$ or *statement*$_{body}$. Therefore, multiple task request handlers and undo-redo clauses may be available at the same time as in Figures 5 and 6. Each worker tries to detect a task request by polling at every `do_two` or parallel `for` statement. When the worker detects a task request, it performs temporary backtracking in order to spawn a larger task by invoking as old a handler as possible. If there are undo-redo clauses on the backtracking path, undo clauses are executed in turn for the backtracking and redo clauses are executed in turn for the resumption.

4.2.4 Tascell Programming

We can write the Tascell program in Figure 8 by (1) starting with the C program in Figure 1, (2) adding the keyword `worker` to the procedure `fib`, (3) finding two statements that can be executed in parallel, (4) forming a `do_two` statement with the consideration of the name and structure of the spawned task, and (5) defining the structure and body of the task.

We can write the Tascell program in Figure 9 by starting with Figure 2 as above, except that (1) we find iterations that can be executed in parallel if separate workspaces are supplied, (2) we form a parallel `for` statement, (3) we prepare some workspace in the task structure and adjust the access to it, (4) we form a `dynamic_wind` statement with existing do/undo operations, and (5) we adjust the parameter and body of `search` in order to accept a task with iterations. Notice that this program avoids undesirable workspace copying and promotes the reuse/sharing of the workspace.

5. Implementation

We implemented a Tascell compiler as a translator to the C language in order to make our implementation portable. It is difficult to realize the backtracking mechanism in "standard" C because it needs "stack walk," accessing variables whose values are located below the current frame in the execution stack. We proposed to implement various language features that require stack walk by using *nested functions*. We applied this scheme to implement Tascell.

5.1 Nested functions

A nested function is a function defined inside another function, in places where variable definitions are allowed except at the top-level. Its evaluation creates a lexical closure accompanying the creation-time environment, and indirect calls to it provide legitimate stack access. Figure 10 shows an example of a program with nested functions. When we (indirectly) call the function `probe_pc1` nested in `fib`, we can access a parameter `probe_pc0` and a local variable `pc` locating in the (older) frame. In this example, by using a chain of nested functions, we can probe `pc` in the `depth`-th newest frame.

```
int fib(int (*probe_pc0) (int), int n)
{
 int pc = 0;
 /* nested function */
 int probe_pc1 (int k){
   if(k == 0) return pc;    /* probe variable pc */
   return probe_pc0(k - 1); /* probe caller's variable */
 }
 if (requested) reply(probe_pc1(depth));
 if (n <= 2) return 1;
 {
   int s1, s2;
   pc = 1; /* inc program counter before call */
   s1 = fib(probe_pc1, n - 1);
   pc = 2; /* inc program counter before call */
   s2 = fib(probe_pc1, n - 2);
   return s1 + s2;
 }
}
```

Figure 10. Program with nested functions.

5.2 Implementations of nested functions

The most well-known implementation of nested functions for C is the *trampoline*-based implementation in GCC [1, 15]. However, maintenance/creation costs of lexical closures in this implementation are high because it performs runtime code generation and prevents local variables and parameters that are accessed by nested functions from being register-allocated.

Therefore, in our previous work, we realized *L-closure*-based implementations of nested functions. These implementations achieve remarkably low maintenance/creation costs by delaying the initialization of the closure until it is invoked and enabling register allocation. We have two versions of L-closure implementations: a translator to standard C (called *LW-SC*) [8] and an enhancement to GCC (called *XC-cube*) [23]. The former allows easy support for various platforms with existing C compilers, and the latter provides a higher performance by using assembly-level implementation techniques.

5.3 Translation to C with nested functions

The program in Figure 8 is translated to the program in Figure 11 with nested functions. Each worker function is translated to have an additional parameter `_bk0` holding a nested function pointer corresponding to the newest handler for a `do_two`, parallel `for`, or `dynamic_wind` statement. Each `do_two` statement is translated into a piece of code that includes a definition of a nested function (`_bk1_do_two` in Figure 11) as the newest handler, which is called when a task request is detected by polling. The nested function first tries to spawn a larger task by calling a nested function (`_bk0`) that corresponds to the second newest handler (which calls another nested function for the third newest handler and so on). Only if a task request still remains, a new task is created and sent to the requester. After sending a task, the worker returns from the nested function and resumes its own computation.

A parallel `for` statement can be translated in the same way (Figure 12), except that, in the nested function, the worker needs to calculate a range for a new task and update a range for itself.

```
int fib(void (*_bk0) (void), struct thread_data *_thr,
        int n)
{
  if (n <= 2)
    return 1;
  else {
    int s1, s2;
    { /*------------------ do_two ------------------*/
      struct tfib pthis[1];    // workspace
      int spawned = 0;         // statement2 is spawned?
      {
        void _bk1_do_two (void) // nested function
        {
          if (spawned) return;
          _bk0(); // continue backtracking
          if (task request exists?) {
            pthis->n = n - 2;      //statementput
            spawned = 1;
            make_and_send_task(_thr, 0, pthis); // spawn
          }
        }
        if (_thr->req)     // polling
          _bk1_do_two (); // start backtracking (call the nested
                          // function defined above)
        {
          s1 = fib(_bk1_do_two, _thr, n-1); // statement1
        }
      }
      if (spawned) {
        // Get and integrate the result of the spawned task
        wait_rslt(_thr);
        s2 = pthis->r;                    // statementget
      } else {
        s2 = fib(_bk0, _thr, n - 2);    // statement2
      }
    } /*------------------ do_two ------------------*/
    return s1 + s2;
  }
}
```

Figure 11. Translation result from the worker function `fib` in Figure 8, including translation of a `do_two` statement.

Translation for a `dynamic_wind` statement is also included in Figure 12. As you can see, *statement*$_{body}$ employs a nested function (`_bk2_dwind` in Figure 12), which is composed of (a copy of) *statement*$_{after}$ (as undo operations), a call to the second newest nested function, and (a copy of) *statement*$_{before}$ (as redo operations), in order to perform undo/redo operations as is described in Section 4.2.3.

6. Discussion

This section compares our approach to related work.

6.1 Multithread-based load balancing

LTC [12] is one of the best implementation techniques for dynamic load balancing. In LTC, a newly spawned logical thread is directly and immediately executed like a usual call while (the continuation of) the oldest thread in the worker may be stolen by another idle worker. Usually, the idle worker (*thief*) randomly selects another worker (*victim*) for stealing a task. Cilk employs this technique.

A message passing implementation [3] 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 created by splitting the present running task. OPA [19], StackThreads/MP [18], and Lazy Threads [6] employ this technique. Polling methods often improve performance by avoiding "memory barrier" instructions, as Indolent Closure Creation [16] improves Cilk's performance.

WorkCrews [21], Leapfrogging [22], and Lazy RPC [4] take the parent-first strategy; at a fork point, a worker executes the parent thread prior to the child thread and makes the child stealable for other workers, and calls the child thread if it has not been stolen at the join point of the parent thread. Tascell uses a similar strategy;

```
int search (void(*_bk0) (void), struct thread_data *_thr,
            int k, int j0, int j1, int j2,
            struct pentomino *tsk)
{
 int s = 0;                  // the number of solutions
 { /*------------------ parallel for ------------------*/
  int p = j1;   int p_end = j2;
  struct pentomino *pthis;
  int spawned = 0;            // the number of spawned tasks
  void _bk1_par_for (void){  // nested function
   if (!spawned) _bk0();      // continue backtracking
   while (p + 1 < p_end && task request exists?) {
    int i1 = (1 + p + p_end)/2,
        i2 = p_end;          // the range for the sub-task
    p_end = i1;              // shrink the range for itself
    pthis = malloc(sizeof(struct pentomino));
                             // allocate a workspace
    { //statementput
      copy_piece_info(pthis->a, tsk->a);
      copy_board(pthis->b, tsk->b);
      pthis->k = k; pthis->i0 = j0;
      pthis->i1 = i1; pthis->i2 = i2;
    }
    spawned++;
    make_and_send_task(_thr, 0, pthis); // spawn
   }
  }
  if (_thr->req)    // polling
   _bk1_par_for(); // start backtracking (call the nested function)
  for (; p < p_end; p++) {
   int ap = tsk->a[p];
   for (each possible direction d of the piece) {
    // examine the ''i-th'' (piece, direction)
    ... 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 ------------*/
     { // do operation (statementbefore)
       Set the ap-th piece onto the board tsk->b and update tsk->a.
     {
      void _bk2_dwind (void) // nested function
      {
       { // undo operation (statementafter)
         Backtrack, i.e., remove the ap-th piece from tsk->b
         and restore tsk->a. }
       _bk1_par_for(); // continue backtracking (call the
                       // nested function defined above)
       { // redo operation (statementbefore)
         Set the ap-th piece onto the board tsk->b
         and update tsk->a. }
      }
      { // statementbody
        kk = the next empty cell;
        if (no empty cell?) s++;     // a solution found
        else  // try the next piece
         s += search (_bk2_dwind, _thr,
                      kk, j0+1, j0+1, 12, tsk);
      }
     }
     { // undo operation (statementafter)
       Backtrack, i.e., remove the ap-th piece from tsk->b
       and restore tsk->a. }
    } /*------------ dynamic_wind ------------*/
   }
  }
  while (spawned-- > 0) {
   // Get and integrate results of spawned tasks
   pthis = (struct pentomino *)wait_rslt(_thr);
   s += pthis->s;          // statementget
   free(pthis); }
 } /*------------------ parallel for ------------------*/
 return s;
}
```

Figure 12. Translation result from the worker function `search` for Pentomino in Figure 9, including translation of a parallel `for` statement and a `dynamic_wind` statement.

however, creations of stealable entities are delayed and mostly omitted.

Lazy Threads [6] realizes further optimization for spawning a thread by translating it into a *parallel ready sequential call*. It achieves a lower thread creation cost than the original LTC by avoiding operations for queueing a new thread. However, this technique can be applied only for consecutive forks. Furthermore, it is unclear how this technique can coexist with the *oldest-first* work stealing strategy.

Our approach is "logical thread"-free, but its ability to restore task-spawnable states without loss of good serial efficiency depends heavily on L-closures and the notion of lazy stack frame management [8, 23]. The idea of lazy frame management can also be applied to logical threads. Indolent Closure Creation [16] employs this idea for Cilk; its technique of using a shadow stack is similar to the lazy validation of an explicit stack in our transformation-based implementation [8] of L-closures. Moreover, our previous work [19] shows that the notion of "laziness" is effective for modern multithreaded languages with thread IDs and dynamically-scoped synchronizers.

We can find few pieces of recent work that make remarkable advances following the abovementioned techniques; for example, X10's thread (or activity) creation and synchronization are inspired by Cilk, and they do not propose a new technique for load balancing [2]. This means that the LTC/Cilk-originating ideas of "logical threads" for load balancing reach maturity.

Notice that our proposal is to employ different semantics from multithreading rather than to reduce costs for multithreading. Our approach enables further performance improvement by reusing a workspace and delaying copying between workspaces. This is the case in most multithreaded languages other than Cilk. In Cilk, a pseudovariable `SYNCHED` is provided, which promotes the reuse of a workspace among child logical threads [17]; however, child threads cannot share a workspace with their parent thread.

Except for not using a "logical thread," our usage of multiple workers is quite usual; thus, our framework can be enhanced with existing/new techniques proposed in previous/future work for other aspects of parallel computing, such as duplicate elimination (especially, in search algorithms) and efficient data placement.

6.2 Distributed memory environment support

Tascell supports distributed memory environments (including mixed-endian environments) by transmitting inputs and outputs as serialized task objects among computation nodes. This support works well if the task size (work amount) is large enough to make the communication cost relatively low. Programmers can write a single program for both shared and distributed memory environments because the interface for passing task objects is integrated. Furthermore, it is easy for new computation nodes to join a running computation dynamically.

Distributed Cilk [14] and SilkRoad [13] employ DSM (Distributed Shared Memory) to support distributed memory environments. DSM is useful to support globally shared data. For this purpose, we may also employ additional libraries or language support.

6.3 Productivity

Tascell provides high productivity in the sense that we can write a Tascell program by augmenting an existing C program, as described in Section 4.2.4. The resulting program would be much simpler than library-based parallelizing frameworks such as TBB [9]. However, when compared to Cilk programs, Tascell programs are more verbose; we need to define `task`s and write statements for task inputs/outputs. These costs are necessary for (1) (general) distributed memory environment support and (2) more exact control of workspaces in task objects with and without `dynamic_wind`.

	Tascell server	computation node (Tascell/Cilk)
CPU	AMD Opteron 244 1.8GHz	AMD Dual Core Opteron 265 1.8GHz×2 (4 cores in total)
OS	Rocks 4.0 (Linux kernel 2.6.9)	
compiler	Allegro Common Lisp 8.1 with `(speed 3) (safety 1) (space 1)` optimizers	GCC 3.4.3 with `-O2` optimizers Tascell compiler + LW-SC (in Tascell) Cilk version 5.3 (in Cilk)
worker	—	created by `pthread_create` with `PTHREAD_SCOPE_SYSTEM`
internode communication (Tascell)	Each computation node is TCP/IP connected to the single Tascell server on Gigabit Ethernet. A node having a task does not send a "to any" task request to another node.	

Table 1. Evaluation environment.

Of course, we may use a more concise description if we limit our language to support only shared memory environments or some limited patterns of parallel computation.

7. Evaluation

In this section, we evaluate the performance of the Tascell framework using the following programs:

Fib(n) recursively computes the n-th Fibonacci number.

Nqueens(n) finds all solutions to the n-queens problem.

Pentomino(n) finds all solutions to the Pentomino problem with n pieces (using additional pieces and an expanded board for $n > 12$).

LU(n) computes the LU decomposition of an $n \times n$ matrix.

Comp(n) compares array elements a_i and b_j for all $0 \le i, j < n$.

Grav(n) computes a total force exerted by $(2n + 1)^3$ uniform particles.

Nqueens in Tascell is coded with a combination of a parallel `for` and a `dynamic_wind` in the same way as Pentomino. LU and Comp use cache-oblivious recursive algorithms with `do_two` constructs. A Comp task to compare arrays of size n and m ($n \ge m$) is divided into two tasks with arrays of $n/2$ and m. Grav is an iterative application. It is implemented in Tascell as triply nested parallel `for` loops corresponding to three axes. Note that we used fine-grained implementations for these applications.

The evaluation environment is summarized in Table 1.

In order to evaluate serial overheads, we ran the Tascell programs with one worker and compared their execution time with C and Cilk programs in almost the same algorithms. For Nqueens, Pentomino, and Grav in Cilk, each thread requires its own workspace to hold one or more arrays. Furthermore, for Nqueens and Pentomino, each thread needs its own copy of its parent thread's workspace even when `SYNCHED` is used, resulting in considerable copying overhead. In Tascell, the worker can reuse a single workspace while it performs a sequential computation as is shown in Section 3.

The results of the performance measurements are shown in Table 2. The overheads in Tascell, which arise from polling and managing nested functions, are considerably lower than Cilk for almost all applications. In particular, Fib shows a sharp contrast in overheads because the frequent creation of logical threads causes a higher overhead in Cilk. Nqueens shows a higher contrast than Pentomino because of more frequent copying. LU shows virtually

	Elapsed time in seconds (relative time to plain C)				
	C	Cilk	Cilk w/o `SYNCHED`	Tascell	Tascell w/ copying
Fib(40)	0.926 (1.00)	7.15 (7.72)	—	2.30 (2.48)	—
Nqueens(15)	10.3 (1.00)	29.8 (2.89)	37.1 (3.60)	15.8 (1.53)	25.4 (2.47)
Pentomino(12)	1.68 (1.00)	2.37 (1.41)	2.74 (1.63)	2.19 (1.30)	2.80 (1.67)
LU(2000)	8.66 (1.00)	8.54 (0.986)	—	8.69 (1.00)	—
Comp(30000)	4.31 (1.00)	7.84 (1.82)	—	5.42 (1.26)	—
Grav(200)	3.35 (1.00)	7.01 (2.09)	—	4.55 (1.35)	—

Table 2. Execution time (and relative time to sequential C programs) with one worker. In Nqueens and Pentomino of Cilk, we partially avoided needless allocation of workspaces by using `SYNCHED`. We used `SYNCHED` only for avoiding needless allocation; initialization of workspaces was always done by copying between workspaces. In "w/o `SYNCHED`", `SYNCHED` was not used; allocation was performed for every spawned logical thread. In "w/ copying" of Tascell, we performed artificial workspace allocation and copying between workspaces for each spawnable task.

zero overheads in both Cilk and Tascell because the potential task division is infrequent.

The additional overheads in Cilk can be broken down as follows:

(a) cost of explicit frame management,

(b) cost of the *THE* protocol [5] for consistent access to the logical thread queue, and

(c) cost of copying between workspaces for each thread (for Pentomino and Nqueens).

The copying overhead can be estimated as the difference between Tascell programs with and without the artificial copying shown in Table 2. The effect of reusing allocated workspaces in Cilk with `SYNCHED` can be estimated as the difference shown in Table 2.

Figure 13 summarizes the results of performance measurements with multiple workers in a shared memory environment. In all benchmarks except LU, Tascell shows higher efficiency (see Figure 13's caption) than Cilk because of Tascell's lower serial overheads. For instance, we achieved a speedup of 1.86 times ($= 0.692/0.372$) as compared with Cilk in Nqueens(16) with 4 workers. Tascell's relative efficiency degradation in Fib with multiple workers is larger than Cilk's because Tascell's overheads for intranode communication are higher, although Tascell's absolute efficiency is considerably higher than Cilk's. The sudden efficiency drop in LU and Grav with 4 workers may be caused by memory bandwidth saturation.

The logarithmically scaled graphs in Figure 14 show the results of performance measurements on multiple computation nodes. [3]

When a single worker is running in each node (Figure 14 (a)), the programs except LU exhibit good speedups because their computation times are sufficiently long relative to the communication costs among computation nodes for the small numbers of spawned tasks; that is, the potential bottleneck around the Tascell server is insignificant in most applications. In contrast, we could not obtain the speedups in LU. To the best of our knowledge, it is diffi-

[3] We evaluated only Tascell because the standard implementation of Cilk only supports shared memory environments.

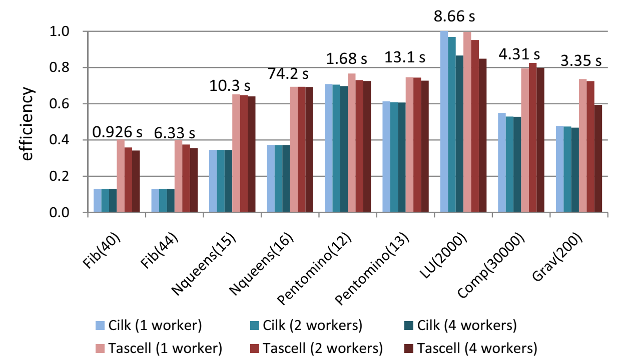

Figure 13. Efficiency with multiple workers in a shared memory environment within one node. Efficiency is defined as S/n_w where S is a speedup to a sequential C program and n_w is the number of workers. (Efficiency $= 1$ means an ideal speedup.) The number above each group of bars shows the execution time in C.

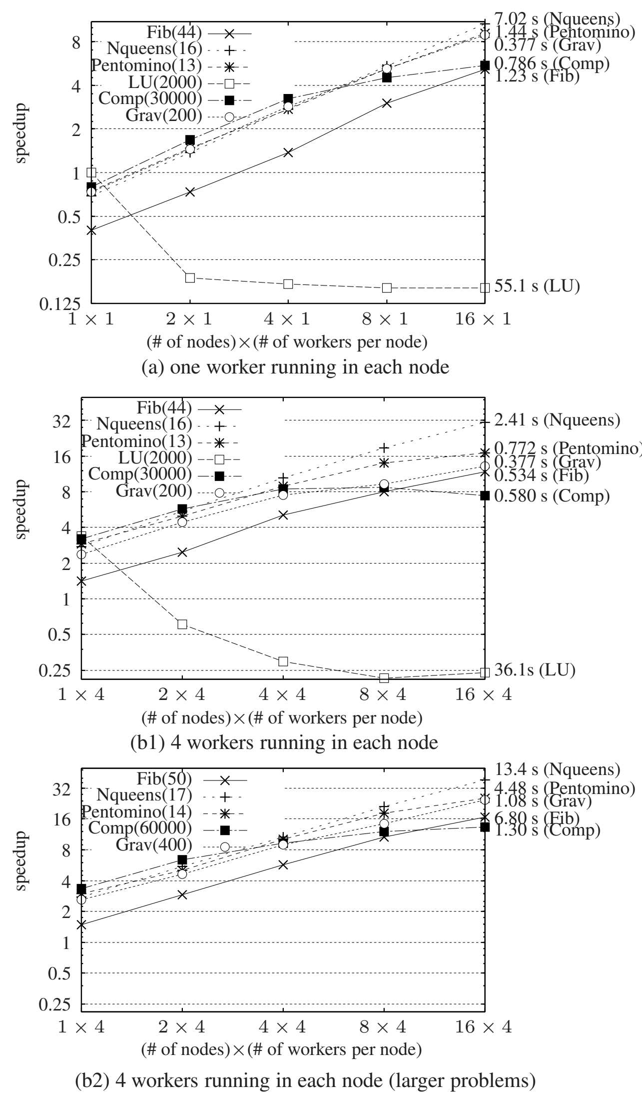

Figure 14. Speedups with multiple computation nodes (relative to sequential C). The numbers on the right side of each graph show execution time with 16 nodes.

cult to obtain sufficient speedups in applications with large shared data such as LU with work-stealing-based dynamic load balancing without efficient support for (distributed) shared memory, as experienced also in [20], since dynamically stolen (transmitted) tasks/results must involve large submatrices.

In environments where multiple workers are running in each node (Figure 14 (b1), (b2)), the workers run with both internode and intranode communication. Figure 14 (b2) shows that we can get a good speedup also in such an environment as long as the size of a problem is sufficiently large relative to the number of workers (e.g., > 4s for each worker). The speedups in Comp are limited because transmission costs of $O(n)$ do not pay for small n since the time complexity of Comp is $O(n^2)$.

We can improve the performance in distributed memory environments by improving the message handling of a Tascell server or employing some mechanism for sharing data among computation nodes.

8. Conclusion and Future work

We proposed a new scheme for dynamic load balancing on the basis of backtracking. Our scheme is useful especially for backtrack search algorithms where overheads are strongly reduced by delayed copying between workspaces, and we can write such algorithms elegantly. In addition, our scheme enables many applications to run more efficiently by allocating workspaces lazily and eliminating the cost of creating/managing logical threads. Furthermore, our task-object-based parallel programming model enables programs to be easily written and executed for both shared and distributed (and also hybrid) memory environments.

We will try to improve the performance in distributed memory environments by implementing more sophisticated message handling among computation nodes or lazy and/or asynchronous data transmission, which will also alleviate potential bottlenecks around Tascell servers. We will also implement a mechanism to enable computation nodes to leave safely.

Acknowledgments

This work was partly supported by Grant-in-Aid for JSPS Fellows (192782) and MEXT Grant-in-Aid for Exploratory Research (17650008).

References

[1] Thomas M. Breuel. Lexical closures for C++. In *Usenix Proceedings, C++ Conference*, 1988.

[2] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, 2005. ISSN 0362-1340.

[3] Marc Feeley. A message passing implementation of lazy task creation. In *Proceedings of the International Workshop on Parallel Symbolic Computing: Languages, Systems, and Applications*, number 748 in Lecture Notes in Computer Science, pages 94–107. Springer-Verlag, 1993.

[4] Marc Feeley. Lazy remote procedure call and its implementation in a parallel variant of C. In *Proceedings of International Workshop on Parallel Symbolic Languages and Systems*, number 1068 in Lecture Notes in Computer Science, pages 3–21. Springer-Verlag, 1995.

[5] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. *ACM SIGPLAN Notices (PLDI '98)*, 33(5):212–223, 1998.

[6] Seth C. Goldstein, Klaus E. Schauser, and David E. Culler. Lazy Threads: Implementing a fast parallel call. *Journal of Parallel and Distributed Computing*, 3(1):5–20, August 1996.

[7] Robert H. Halstead, Jr. New ideas in parallel Lisp: Language design, implementation, and programming tools. In T. Ito and R. H. Halstead, editors, *Parallel Lisp: Languages and Systems*, volume 441 of *Lecture Notes in Computer Science*, pages 2–57, Sendai, Japan, June 5–8, 1990. Springer, Berlin.

[8] Tasuku Hiraishi, Masahiro Yasugi, and Taiichi Yuasa. A transformation-based implementation of lightweight nested functions. *IPSJ Digital Courier*, 2:262–279, 2006. (IPSJ Transaction on Programming, Vol. 47, No. SIG 6(PRO 29), pp. 50-67.).

[9] Intel Corporation. *Intel Threading Building Block Tutorial*, 2007. `http://threadingbuildingblocks.org/`.

[10] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9): 26–76, September 1998.

[11] B.C. Kuszmaul. Cilk provides the "best overall productivity" for high performance computing:(and won the HPC challenge award to prove it). *Proceedings of the nineteenth annual ACM Symposium on Parallel Algorithms and Architectures*, pages 299–300, 2007.

[12] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3): 264–280, July 1991.

[13] Liang Peng, Weng Fai Wong, Ming Dong Feng, and Chung Kwong Yuen. SilkRoad: A multithreaded runtime system with software distributed shared memory for SMP clusters. In *IEEE International Conferrence on Cluster Computing (Cluster2000)*, pages 243–249, November 2000.

[14] K. Randall. Cilk: Efficient multithreaded computing. Technical Report MIT/LCS/TR-749, 1998.

[15] R. M. Stallman. Using and porting GNU Compiler Collection. 1999.

[16] Volker Strumpen. Indolent closure creation. Technical Report MIT-LCS-TM-580, MIT, June 1998.

[17] Supercomputing Technologies Group. *Cilk 5.4.6 Reference Manual*. Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, Massachusetts, USA.

[18] Kenjiro Taura, Kunio Tabata, and Akinori Yonezawa. StackThreads/MP: Integrating futures into calling standards. In *Proceedings of ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP'99)*, pages 60–71, May 1999.

[19] Seiji Umatani, Masahiro Yasugi, Tsuneyasu Komiya, and Taiichi Yuasa. Pursuing laziness for efficient implementation of modern multithreaded languages. In *Proceedings of the Fifth International Symposium on High Performance Computing*, number 2858 in Lecture Notes in Computer Science, pages 174–188, October 2003.

[20] Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. In *In Proceedings of the eighth ACM SIGPLAN symposium on Principles and Practices of Parallel Programming (PPoPP'01)*, pages 34–43, New York, NY, USA, 2001. ACM. ISBN 1-58113-346-4.

[21] Mark T. Vandevoorde and Eric S. Roberts. WorkCrews: An abstraction for controlling parallelism. *International Journal of Parallel Programming*, 17(4):347–366, 1988.

[22] David B. Wagner and Bradley G. Calder. Leapfrogging: A portable technique for implementing efficient futures. In *Proceedings of Principles and Practice of Parallel Programming (PPoPP'93)*, pages 208–217, 1993.

[23] Masahiro Yasugi, Tasuku Hiraishi, and Taiichi Yuasa. Lightweight lexical closures for legitimate execution stack access. In *Proceedings of the Fifteenth International Conference on Compiler Construction (CC2006)*, number 3923 in Lecture Notes in Computer Science, pages 170–184. Springer-Verlag, 2006.