On Efficient Load Balancing for Irregular Applications

Masahiro Yasugi

Department of Artificial Intelligence, Kyushu Institute of Technology,
680-4 Kawazu Iizuka Fukuoka, Japan 820-8502

Abstract. This short essay overviews a history and a future perspective of dynamic load balancing for irregular applications. Since I write this essay for the Festschrift, I discuss ideas of load balancing from the point of view of concurrent objects as much as possible.

Keywords: load balancing, concurrent objects.

1 Introduction

In my doctoral work, I designed a concurrent object-oriented programming language ABCL/ST (ABCL/Statically Typed) based on ABCL/1 [32, 31] and implemented a concurrent object-oriented language system ABCL/EM-4 [28] for a highly parallel data-driven computer EM-4 [19].

In the computation/programming model for ABCL/1, computation is performed by a collection of autonomous, concurrently active software entities called *concurrent objects*, and the interaction between concurrent objects is performed solely via message passing. More than one concurrent object can become active simultaneously, and more than one message transmission may take place in parallel. Each concurrent object has its own single thread of control, and it may have its own memory, the contents of which can be accessed only by itself. Theoretical foundations for concurrent objects have been established by a series of studies in Actors [1], ABCM [20] (the computation model for ABCL/1), POOL [2], and calculi of asynchronous objects [15].

In a concurrent object-oriented language system ABCL/EM-4, the data-driven nature of EM-4 was well suited for efficient message handling. EM-4 consists of 80 PEs (Processing Elements) and the interconnection network. We can allocate a newly created concurrent object on a PE and exchange hardware-level packets between PEs to realize message passing. Interestingly, at the implementation level, I considered PEs as generalized, universal concurrent objects (i.e., abstract PEs). At present, I would like to call such generalized, universal concurrent objects *workers*, which handle implementation-level messages and change their roles according to ABCL-level concurrent objects. Usually, the abstract system consists of as many workers as there are PEs in the underlying computer system.

I also think that the concurrent object-based computation model is useful for implementing higher-level languages than concurrent object-oriented languages,

G. Agha et al. (Eds.): Yonezawa Festschrift, LNCS 8665, pp. 239–250, 2014.

where workers (as universal concurrent objects) actually run in parallel. It is useful for performance analysis as an execution model, and is useful for the design and verification of protocols (for high-level communication by exchanging messages). Concurrent objects have clear modularity; messages and their types (request/reply) are clear, and each concurrent object clearly has its own single thread of control and its own memory.

An actual computer system may have shared memory. The concurrent object-based computation model does not cover shared memory. Shared memory may serve as both an additional gift and a trouble. However, we can specify most interactions among workers without directly using shared memory; of course, shared memory can be used for implementing the concurrent object-based computation model.

Recently, as high-level languages that employ workers (as universal concurrent objects), I am interested in languages which support dynamic load balancing for irregular applications. In the rest of this short essay, I would like to overview a history and a future perspective of dynamic load balancing.

2 Load Balancing

With the growing popularity of parallel architectures including many/multi-cores, it has become important to ensure easy parallel programming for efficient parallel computing. An ultimate goal of programming/computing system research is to allow users to describe the computation at a higher level of abstraction and to automatically determine the details of how to perform the computation.

In many irregular applications, *static* partitioning of work with sufficient concurrency into parallel tasks (for present workers), each with an equal amount of work, is impossible. In such cases, dynamic load balancing, where a task (a piece of work) is dynamically allocated to an idle worker, is effective. Work stealing is a randomized technique that implements load balancing. An idle worker (thief) steals a task from another randomly selected loaded worker (victim). Usually, the number of workers does not exceed the number of underlying computing resources, such as cores and "hardware" threads (afforded by Simultaneous Multi-threading (SMT) and Hyper-threading), so that workers actually run in parallel.

In general, work-stealing frameworks [26, 18, 27, 6, 7, 3, 8, 21, 23, 25, 16, 4, 5, 17, 13, 10, 11] work well with parallel divide-and-conquer (tree-recursive) algorithms, where workers, if necessary, exchange relatively large subdivided tasks near the root of the invocation tree in order to reduce the total work-stealing costs. It is well known that (nested) FORALL-style parallel algorithms can be converted into parallel divide-and-conquer (recursive) algorithms easily (almost automatically); for example, TBB [16]'s `parallel_for` template function recursively splits a given range into subranges. Examples of manual conversion for Cilk [8] can be found in our previous paper [29].

Work-stealing frameworks with parallel divide-and-conquer algorithms generally afford better cache locality than *work-sharing* approaches; when the `dynamic`

schedule is specified on OpenMP's work-sharing loop construct, relatively small chunks of iterations are assigned to workers (called "threads in the team" in OpenMP) dynamically. Note that work-sharing with larger chunks (even with the `guided` schedule) is less tolerant of work imbalance among iterations.

2.1 Lazy Task Creation

LTC (Lazy Task Creation) [18] is one of the best implementation techniques for dynamic load balancing. LTC 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.

In LTC, each worker spawns plenty of logical threads and schedules them internally and thus efficiently. An idle worker (thief) may steal (the continuation of) 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 oldest-first work-stealing strategy is generally effective in making tasks larger.

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 [8] employs this technique. LTC is originally invented for MultiLisp [12], where the *future* construct is used as in `(+ (future` E_1`) (future` E_2`))`. A *future* expression creates a logical thread, and the channel (or promise) of the result (which will be determined later) is passed to the continuation of the *future* expression. The result is waited for and extracted with an implicit *touch* operation.

A message passing implementation [6] of LTC employs a polling method where the victim detects a task request sent by the thief and returns a new task created by splitting the present running task. StackThreads/MP [23], OPA [25], and Lazy Threads [9] employ this technique. Although the thief may have to wait for a task for a long time, polling methods often improve performance by avoiding "memory barrier" instructions, as Indolent Closure Creation [21] improves Cilk's performance.

2.2 Cilk

Cilk [8] is a parallel programming language. It provides good load balancing by employing LTC [18].

In Cilk, the programmer specifies parallel functions (`cilk` procedures). The spawning of a parallel function is written as a C call with an additional `spawn` keyword. At the language level, a logical thread that executes the parallel function is created. At the implementation level, this child thread is executed immediately (prior to the parent), and (the continuation of) the parent thread becomes stealable for dynamic load balancing. The programmer writes a `sync` statement so that the parent thread waits for the completion of all spawned child threads. Note that `sync` statements are compiled away for *fast clones* [22] at the

implementation level. Since each parallel function has `sync` as its implicit last statement, the child threads cannot survive longer than the parent thread. Thus, the termination of a parallel algorithm is simply detected as the completion of the corresponding parallel function invocation.

Note that Cilk can run on shared memory environments, but it cannot run on distributed memory environments. Since the continuation of the parent thread is implicit, explicit serialization/communication (for distributed memory) of the implicit continuation context is difficult.

Cilk employs a Dijkstra-like (and Dekker-like) protocol called the "THE" protocol for work stealing. When this protocol is implemented on modern parallel architectures that do not provide sequential consistency for shared memory, the owner (the potential victim) is forced to execute store-load memory barrier (fence) instructions when extracting its own potential tasks (logical threads in Cilk); this results in substantial overheads.

In Cilk, the pseudovariable `SYNCHED` is true if all spawned child threads are completed. `SYNCHED` was originally introduced to promote the reuse of a workspace[1] among child logical threads [22] and usually it cannot be used for the reuse of a workspace between parent and child logical threads.

2.3 Other Versions of Cilk

In 1995, a previous version of Cilk was published in [3]. This version of Cilk differs from the well-known version Cilk-5 [8] in the following manner:

- Each logical thread is nonblocking, which means that it can run to completion without waiting or suspending once it has been invoked.
- Programs must be written in explicit continuation-passing style; each logical thread must additionally spawn a successor thread (by `spawn_next`) to receive the children's return values when they are produced.
- When a thread spawns a new child thread, the parent-first approach is taken; i.e., the new child thread (rather than the parent thread) is pushed to the ready queue. Later, it will be popped or stolen. (More precisely, they use *levels* each of which corresponds to the number of `spawn`'s (but not `spawn_next`'s) to employ the ready queue as an array in which the Lth element contains a linked list of all ready threads having level L.)
- It can run on distributed memory computers. When threads are stealable, they have not been invoked and they have no continuation contexts. Implicit continuations in Cilk-5 may be expressed explicitly as such stealable new threads in [3]. Note that a thread may spawn as many threads as necessary, and creation of implicit Cilk-5 intrathread continuations can be mostly avoided.

Recently, MIT licensed Cilk technology to Cilk Arts, Inc. Cilk Arts developed Cilk++, which includes full support for C++, parallel loops, and superior interoperability with serial code. In July 2009, Intel Corporation acquired Cilk Arts.

[1] In this essay, *workspaces* mean arrays or any other mutable data structures.

Intel released its ICC compiler with Intel Cilk Plus for C and C++ and provided the GCC "cilkplus" branch C/C++ compiler.

In Cilk++ (and Cilk Plus), reducers and other Cilk++ hyperobjects are introduced, which enable lazy allocation of views (race-free reduction workspaces). That is, when the continuation of the parent thread is stolen, a new view of the reducer is allocated. This behavior can also be implemented with `SYNCED` in Cilk-5, but the use of reducers is easy to understand.

2.4 Tascell

Recently, we proposed a "logical thread"-free parallel programming/execution framework called *Tascell* as an efficient work-stealing framework [13]. Tascell implements *backtracking-based load balancing* with on-demand concurrency. A worker performs a computation sequentially unless it receives a task request with polling. When requested, the worker spawns a "real" task by temporarily "backtracking" and restoring its oldest task-spawnable state. Because no *logical threads* are created as potential tasks, the cost of managing a queue for them can be eliminated.[2] Tascell also promotes the long-term (re)use of workspaces (such as arrays and other mutable data structures) and improves the locality of reference since it does not have to prepare a workspace for each concurrently runnable logical thread.

The Tascell framework consists of a compiler for the Tascell language and a runtime system. This framework supports both distributed and shared memory environments. The Tascell compiler employs an extended C language as the intermediate language. In the first compilation phase, a Tascell program is translated into an extended C program with nested function definitions in order to implement *task-request handlers* and *dynamic winders*. In the second compilation phase, the extended C program with nested functions is compiled by an enhanced version of GCC [30] or by a translator into standard C [14]. These implementations provide lightweight lexical closures called "L-closures" which are created by evaluating nested function definitions, enabling a running program to legitimately inspect/modify the contents of its execution stack. Using elaborate compilation/translation techniques, we can delay the initialization of an L-closure until the L-closure is actually invoked, and we can use a private location as a register allocation candidate for an accessed variable to realize quite low creation/maintenance costs. Because the compiled Tascell program creates L-closures very frequently but calls them infrequently (only when spawning a task), the total overhead can be reduced significantly even with high invocation costs.

Idle workers request tasks from *loaded* workers. When receiving a task request, a loaded worker (victim) creates a new task by dividing the current running task,

[2] The effect is significant only when the cost of managing logical threads is relatively high (in expected time) as in Cilk-5. There are multithreaded languages and systems (such as StackThreads/MP [23] and OPA [25]) in which the cost of managing logical threads is quite low.

and returns the new task to the idle worker. When an idle worker (thief) receives a task, it executes the task and returns the result of the task.

In the current implementation of Tascell, each worker employs a single execution stack for multiple tasks. When a worker must wait for the result of a stolen task, it calls a C function which attempts to steal (and execute with the worker's stack) another task by dividing the stolen task.[3] When the result is available, the return to the "current" task is performed as the ordinary return in C; that is, the "current" task is managed (or "suspended") with C's call/return mechanism.

In Tascell, spawned tasks are managed. More precisely, a task request, the stolen (spawned) task, the result of the stolen task, and the ACK of the result are managed by both the victim and thief workers. A task and its result are represented by a task object.

Figure 1 shows a Tascell program that performs backtrack search for finding all possible solutions to the Pentomino puzzle. We defined a task object named `pentomino`. Several fields are declared as the search input. The field `s` is declared for storing the result. A Tascell worker that receives a `pentomino` task executes `pentomino`'s `task_exec` body. In the `task_exec` body, Tascell worker can refer to the received task object by the keyword `this`.

In *worker functions*, which are specified by the keyword `worker` (like `cilk` procedures in Cilk), we can use Tascell's task division constructs. A parallel `for` loop construct can be used for dividing an iterative computation. It is syntactically denoted by:

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

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, whose object is initialized by $statement_{put}$. In $statement_{put}$, the actual assigned range can be referred to by $identifier_{from}$ and $identifier_{to}$. The worker handles the result of the spawned task by executing $statement_{get}$.[4] Note that a worker performs iterations for a parallel `for` loop sequentially unless requested; the worker does not create any logical threads and can (re)use a single workspace (such as a worker-local array) for a long time.

Parallel `for` statements may be nested *dynamically* in their $statement_{body}$. Therefore, multiple task-request handlers may be available at the same time.

[3] This saves the execution stack as in Leapfrogging [27]. TBB [16] employs a more general technique for saving the execution stack.

[4] Specifying a task definition and several statements to handle task objects makes Tascell programs more verbose than Cilk programs. These costs are necessary for more exact control of workspaces and distributed memory environment support.

```
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. 1. A Tascell program that performs backtrack search for Pentomino

Each worker attempts to detect a task request by polling at every parallel `for` statement without heavy memory barrier (fence) instructions. 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.

We may use the `dynamic_wind` construct in order to specify how to perform undo-redo operations during the backtracking (undo) and the return from the backtracking (redo). In Figure 1, the worker employs a single workspace for representing a board with pieces (within `tsk`) unless it receives a task request; the *dynamic winder* temporarily removes pieces in order to restore a task-spawnable state near the root of the backtrack search tree so that the oldest *task-request handler* can spawn a larger task as `this` by copying the restored workspace (within `tsk`).

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 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.

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 [14, 30]. The idea of lazy frame management can also be applied to logical threads. Indolent Closure Creation [21] 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 [14] of L-closures. StackThreads/MP [23] enables each worker to manage logical threads within its execution stack by allowing the frame pointer to walk the execution stack independently of the stack pointer. (Our compiler-based implementation [30] of L-closures is based on generalization of this technique.) Moreover, our previous work [25] shows that the notion of "laziness" is effective

for modern multithreaded languages with thread IDs and dynamically-scoped synchronizers.

Notice that Tascell's approach is to employ different semantics from multithreading rather than to reduce costs for multithreading. Tascell's 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 [22].

2.5 Other Frameworks

WorkCrews [26], Leapfrogging [27], and Lazy RPC [7] 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; however, creations of stealable entities are delayed and mostly omitted.

Lazy Threads [9] 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.

We can find few pieces of recent work that make remarkable advances following the abovementioned techniques; for example, X10[4]'s thread (or activity) creation and synchronization are inspired by Cilk, and the fundamental parts of recent techniques [5, 17, 10, 11] are not beyond the abovementioned techniques. This means that the LTC/Cilk-originating ideas of "logical threads" for load balancing reach maturity.

3 Future Perspective

There is a research topic in Tascell. In Tascell, when a worker waits for the result of a stolen task, it tries to steal (and executes) another task of the task requester until the result is returned. This restriction is posed for saving the execution stack as in Leapfrogging [27]. This also limits the choice of tasks to steal and therefore might limit parallelism and cause tightly stealing workers. The use of multiple execution stacks for a single worker would alleviate the problem. In addition, the only temporary use of a constantly bounded execution stack would solve the problem; this means that continuations should be stealable.

For multithreaded languages (and Tascell with stealable continuations), the implementation for distributed memory is difficult mainly because the implicit stealable continuations are difficult to move to another node. The idea of Cilk++ hyperobjects, where the objects themselves can interact with steals, may be applied to this problem.

With the growing popularity of highly/massively parallel architectures, scalability and dependability will be very important properties of load balancing

frameworks. Extending load balancing frameworks to support these properties would be required.

In a large scale shared memory system, preventing data races would become important. Type systems may be useful for preventing data races from occurring in the first place (e.g., based on [24]).

Recent computer architectures for high performance computing tend to exploit heterogeneity and hierarchy. Software systems also employ multiple programming languages and advanced features such as dynamic compilation. Load balancing frameworks and other frameworks should exploit these aspects for efficiency, meaning not only high performance but also low energy consumption. Clear modularity of concurrent objects can provide clear (extended) models for the design and verification of such complex systems.

References

1. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. The MIT Press (1987)
2. America, P., Rutten, J.: A layered semantics for a parallel object-oriented languages. In: de Bakker, J.W., de Roever, W.-P., Rozenberg, G. (eds.) REX 1990. LNCS, vol. 489, pp. 91–123. Springer, Heidelberg (1991)
3. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. In: Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 1995, pp. 207–216 (1995)
4. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. SIGPLAN Not. 40(10), 519–538 (2005)
5. Cong, G., Kodali, S., Krishnamoorthy, S., Lea, D., Saraswat, V., Wen, T.: Solving large, irregular graph problems using adaptive work-stealing. In: ICPP 2008: Proceedings of the 2008 37th International Conference on Parallel Processing, pp. 536–545. IEEE Computer Society (2008)
6. Feeley, M.: A message passing implementation of lazy task creation. In: Halstead, R.H., Ito, T. (eds.) US/Japan WS 1992. LNCS, vol. 748, pp. 94–107. Springer, Heidelberg (1993)
7. Feeley, M.: Lazy remote procedure call and its implementation in a parallel variant of C. In: Queinnec, C., Halstead, R.H., Ito, T. (eds.) PSLS 1995. LNCS, vol. 1068, pp. 3–21. Springer, Heidelberg (1996)
8. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. ACM SIGPLAN Notices (PLDI 1998) 33(5), 212–223 (1998)
9. Goldstein, S.C., Schauser, K.E., Culler, D.E.: Lazy Threads: Implementing a fast parallel call. Journal of Parallel and Distributed Computing 3(1), 5–20 (1996)
10. Guo, Y., Barik, R., Raman, R., Sarkar, V.: Work-first and help-first scheduling policies for async-finish task parallelism. In: 23rd IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2009), pp. 1–12 (May 2009)
11. Guo, Y., Zhao, J., Cave, V., Sarkar, V.: Slaw: a scalable locality-aware adaptive work-stealing scheduler. In: 24th IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2010), pp. 1–12 (April 2010)

12. Halstead, R.H.: New ideas in parallel Lisp: Language design, implementation, and programming tools. In: Ito, T., Halstead, R.H. (eds.) US/Japan WS 1989. LNCS, vol. 441, pp. 2–57. Springer, Heidelberg (1990)
13. Hiraishi, T., Yasugi, M., Umatani, S., Yuasa, T.: Backtracking-based load balancing. In: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2009), pp. 55–64 (February 2009)
14. Hiraishi, T., Yasugi, M., Yuasa, T.: A transformation-based implementation of lightweight nested functions. IPSJ Digital Courier 2, 262–279 (2006), IPSJ Transactions on Programming 47(SIG 6(PRO 29)), 50–67
15. Honda, K., Tokoro, M.: An object calculus for asynchronous communication. In: America, P. (ed.) ECOOP 1991. LNCS, vol. 512, pp. 133–147. Springer, Heidelberg (1991)
16. Intel Corporation: Intel Threading Building Block Tutorial (2007), `http://threadingbuildingblocks.org/`
17. Michael, M.M., Vechev, M.T., Saraswat, V.A.: Idempotent work stealing. In: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2009), pp. 45–54 (February 2009)
18. Mohr, E., Kranz, D.A., Halstead, R.H.: Lazy task creation: A technique for increasing the granularity of parallel programs. IEEE Transactions on Parallel and Distributed Systems 2(3), 264–280 (1991)
19. Sakai, S., Yamaguchi, Y., Hiraki, K., Kodama, Y., Yuba, T.: An architecture of a dataflow single chip processor. In: Proc. of the 16th Annual International Symposium on Computer Architecture, pp. 46–53 (June 1989)
20. Shibayama, E.: An Object-Based Approach to Modeling Concurrent Systems. Ph.D. thesis, Department of Information Science, The University of Tokyo (1991)
21. Strumpen, V.: Indolent closure creation. Tech. Rep. MIT-LCS-TM-580, MIT (June 1998)
22. Supercomputing Technologies Group: Cilk 5.4.6 Reference Manual. Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, Massachusetts, USA
23. Taura, K., Tabata, K., Yonezawa, A.: StackThreads/MP: Integrating futures into calling standards. In: Proceedings of ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP 1999), pp. 60–71 (May 1999)
24. Terauchi, T.: Checking race freedom via linear programming. In: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2008, pp. 1–10 (2008)
25. Umatani, S., Yasugi, M., Komiya, T., Yuasa, T.: Pursuing laziness for efficient implementation of modern multithreaded languages. In: Veidenbaum, A., Joe, K., Amano, H., Aiso, H. (eds.) ISHPC 2003. LNCS, vol. 2858, pp. 174–188. Springer, Heidelberg (2003)
26. Vandevoorde, M.T., Roberts, E.S.: WorkCrews: An abstraction for controlling parallelism. International Journal of Parallel Programming 17(4), 347–366 (1988)
27. Wagner, D.B., Calder, B.G.: Leapfrogging: A portable technique for implementing efficient futures. In: Proceedings of Principles and Practice of Parallel Programming (PPoPP 1993), pp. 208–217 (1993)
28. Yasugi, M.: A concurrent object-oriented programming language system for highly parallel data-driven computers and its applications. Tech. Rep. 94-7e, Department of Information Science, Faculty of Science, University of Tokyo (April 1994), Doctoral Thesis (March 1994)

29. Yasugi, M., Hiraishi, T., Umatani, S., Yuasa, T.: Parallel graph traversals using work-stealing frameworks for many-core platforms. Journal of Information Processing 20(1), 128–139 (2012)
30. Yasugi, M., Hiraishi, T., Yuasa, T.: Lightweight lexical closures for legitimate execution stack access. In: Mycroft, A., Zeller, A. (eds.) CC 2006. LNCS, vol. 3923, pp. 170–184. Springer, Heidelberg (2006)
31. Yonezawa, A. (ed.): ABCL: An Object-Oriented Concurrent System — Theory, Language, Programming, Implementation and Application. The MIT Press (1990)
32. Yonezawa, A., Briot, J.P., Shibayama, E.: Object-oriented concurrent programming in ABCL/1. In: Proc. of ACM Conference on OOPSLA, pp. 258–268 (1986)