HOPE: A Parallel Execution Model Based on Hierarchical Omission

Masahiro Yasugi
Kyushu Institute of Technology
Iizuka, Fukuoka, Japan
yasugi@ai.kyutech.ac.jp

Daisuke Muraoka*
Kyushu Institute of Technology
Iizuka, Fukuoka, Japan

Tasuku Hiraishi
Kyoto University
Kyoto, Japan
tasuku@media.kyoto-u.ac.jp

Seiji Umatani
Kanagawa University
Hiratsuka, Kanagawa, Japan

Kento Emoto
Kyushu Institute of Technology
Iizuka, Fukuoka, Japan

ABSTRACT

This paper presents a new approach to fault-tolerant language systems without a single point of failure for irregular parallel applications. Work-stealing frameworks provide good load balancing for many parallel applications, including irregular ones written in a divide-and-conquer style. However, work-stealing frameworks with fault-tolerant features such as checkpointing do not always work well. This paper proposes a completely opposite “work omission” paradigm and its more detailed concept as a “hierarchical omission”-based parallel execution model called HOPE. HOPE programmers’ task is to specify which regions in imperative code can be executed in sequential but arbitrary order and how their partial results can be accessed. HOPE workers spawn no tasks/threads at all; rather, every worker has the entire work of the program with its own planned execution order, and then the workers and the underlying message mediation systems automatically exchange partial results to omit hierarchical subcomputations. Even with fault tolerance, the HOPE framework provides parallel speedups for many parallel applications, including irregular ones.

CCS CONCEPTS

• **Software and its engineering** → **Software fault tolerance**; **Parallel programming languages**.

KEYWORDS

fault tolerance, work omission, parallel execution model, language systems

ACM Reference Format:

Masahiro Yasugi, Daisuke Muraoka, Tasuku Hiraishi, Seiji Umatani, and Kento Emoto. 2019. HOPE: A Parallel Execution Model Based on Hierarchical Omission. In *48th International Conference on Parallel Processing (ICPP 2019)*, August 5–8, 2019, Kyoto, Japan. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3337821.3337899>

*Presently with SmartNews, Inc.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICPP 2019, August 5–8, 2019, Kyoto, Japan

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6295-5/19/08...\$15.00
<https://doi.org/10.1145/3337821.3337899>

1 INTRODUCTION

High productivity, scalability, load balancing, and fault tolerance are all important issues of high-performance and robust applications, including irregular ones, for massively and extremely parallel computing systems.

Work-stealing frameworks [2, 5, 8, 11, 13, 14, 20, 21] have been proposed to provide good load balancing. They work well with parallel divide-and-conquer algorithms, where workers, if necessary, exchange relatively large subdivided tasks near the root of the invocation tree to reduce the total work-stealing costs.

Lazy task creation (LTC) [14] is one of the best implementation techniques for implementing work-stealing frameworks. LTC keeps all workers busy by creating plenty of “logical” threads and adopting the oldest-first work-stealing strategy. In LTC, a newly spawned logical thread is directly and immediately executed like an ordinary call while (the continuation of) the oldest thread in the worker may be stolen by another idle worker. Typically, an idle worker (thief) randomly selects another worker (victim) to steal a logical thread. Cilk [5] employs this technique.

Hiraishi *et al.* proposed a “logical thread”-free parallel programming/execution framework called *Tascell* as an efficient work-stealing framework [8]. *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. Unlike LTC [14] and Cilk [5], the thief steals the newly spawned task (not the oldest thread).¹ Since 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 references because it does not have to prepare a workspace for each concurrently runnable logical thread.

However, work-stealing frameworks enhanced by fault-tolerant features such as checkpointing do not always work well. This is because there is no common global synchronization point among all workers. When a thief steals a newly spawned task from a victim, the thief and the victim share a local synchronization point, and

¹This allows a single *Tascell* program to run in both shared and distributed memory environments.

²The effect is significant when the cost of managing logical threads is relatively high (in expected time) as in Cilk-5 [5]. There are a few multithreaded languages and systems (such as StackThreads/MP [20] and OPA [21]) in which the cost of managing logical threads is remarkably low.

the victim can save the stolen task so that it can restart the same task even when the thief has stopped because of a fault. However, a large portion of a divide-and-conquer computation may be lost around the root. Furthermore, the root is a single point of failure.

A new approach to fault-tolerant language systems without a single point of failure for divide-and-conquer parallel applications is presented herein. This paper proposes a novel “hierarchical work omission”-based parallel execution model called *HOPE*. The key idea of the HOPE model, namely, the “work omission” paradigm may not be a novel idea in regular (data-parallel) applications but is a novel idea for divide-and-conquer parallel applications.

HOPE provides high productivity. The programmers’ task is to hierarchically specify subcomputations and how their results can be accessed; those hierarchical subcomputations are executed sequentially but can be executed in any order. Every worker performs the entire work of the program *sequentially*; that is, it spawns no tasks/threads at all. However, it performs the specified hierarchical subcomputations in its own *planned order*, and may *omit* a hierarchical subcomputation if it knows the result of the subcomputation; workers exchange partial results to achieve parallel speedups. To effectively store and exchange partial results, every worker employs an underlying message mediation subsystem (MMS). MMSs are distributed and federated, each serving as a local storage and an automatic exchanger of messages of partial results.

The contributions of this paper are three-fold:

- This paper presents the fault-tolerant HOPE model. Faults may stop communication among workers and may stop even majority of workers, as long as remaining workers can continue communication with each other (or only within every isolated group) by assuming some reasonable fault resilience in the underlying communication systems [4]. We can handle any faults (e.g., signals, I/O faults, permission errors, memory errors, detection of uncorrectable data, and out of memory errors) if they simply stop the damaged part of the system.
- For scalability, we propose employing pre-shared plans (execution orders) among workers so that the distributed MMSs can adaptively exchange messages of partial results with only relevant MMSs. MMSs also use system messages for good collaboration, such as catch-up and prefetch; catch-up information including multiple messages (partial results) is useful for a new participant of a subcomputation.
- Each worker can check whether the result of a subcomputation is locally available at its underlying MMS. To improve efficiency, we propose adding the “non-check” and “work-first” execution modes. Furthermore, we employ an *on-demand* dynamic execution mode switching technique; for example, the past choice of the “non-check” mode for a subcomputation can be changed later as if it were started in the “check” mode. For this purpose, we use nested functions as extended C languages’ mechanisms for legitimate execution stack access.

Note that HOPE supports as many (irregular and complex) applications as Tascell with fault tolerance by making subcomputations *idempotent*.

The remainder of this paper is organized as follows. We present the backgrounds and motivation in Section 2. We propose our approach in Section 3. In Section 4, we provide a brief description of the HOPE language. We outline the base implementation in Section 5, which will be improved in the following Sections 6 and 7. Section 8 presents that our implementation achieves good performance and fault tolerance. We discuss related work in Section 9. Finally, we conclude this paper in Section 10.

2 MOTIVATION

The standard approach to fault tolerance is to make a system redundant. Instead of executing a program with a single process, we can employ multiple processes in parallel so that at least one will continue execution when the others stop because of faults. This sort of redundant parallel execution is simple and fault-tolerant; however, it does not provide any speedups.

For regular (data-parallel) applications, programs are often transformed into SPMD (single program, multiple data) programs. Each SPMD process manipulates its assigned portion of data (or ranges) and synchronizes/communicates with other parallel SPMD processes for other portions of data (or ranges). As this approach reduces redundancy, we need additional mechanisms to recover fault tolerance.

For simplicity, we assume that we have some mechanisms that use **data redundancy** to recover (e.g., reacquire or recalculate) lost data.

In the case of regular (data-parallel) applications, **control redundancy** can appear as follows: SPMD processes sometimes share a common synchronization point (e.g., for global broadcast/reduction operations); when they observe that some of them have stopped because of faults at the global synchronization point, the remaining SPMD processes can continue execution by manipulating its re-assigned portion of the (recovered) data.

Irregular (task-parallel) applications are important in some fields. They include divide-and-conquer parallel applications such as search problems and applications based on tree-recursive algorithms.

We will use a tree-recursive algorithm as an example. The algorithm is difficult to parallelize while ensuring low overhead and fault tolerance simultaneously; however, we will demonstrate that our approach is effective (even) on it.

The 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, the computations of $\text{fib}(n-1)$ and $\text{fib}(n-2)$ can be executed in parallel. This algorithm has no practical application but is often used (e.g., in [5, 8, 11, 14, 20, 21]) 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.

In Figure 1, we added a right-hand-side comment to easily identify recursive subcomputations with i ; that is, $\text{fib}_i(n)$ recursively contains subcomputations $r0 = \text{fib}_{2i}(n-1)$ and $r1 = \text{fib}_{2i+1}(n-2)$.³ Figure 2 shows the tree of subcomputations in $\text{fib}_1(51)$; for example, s_1 contains s_2 and s_3 , and s_2 contains s_4 and s_5 , and so on.

³Furthermore, we can avoid redundant computations without changing the tree shape if we replace 1_i at line 2 with expressions such as $g(i)$ in Figure 1.

```

1  int fib(int n) {           // int fibi(int n) {
2    if (n <= 2) return 1;    // if (n <= 2) return 1i;
3    int r0, r1;             // int r0, r1;
4    r0 = fib(n - 1);         // r0 = fib2i(n - 1);
5    r1 = fib(n - 2);         // r1 = fib2i+1(n - 2);
6    return r0 + r1;         // return r0 + r1;
7  }                         // }

```

Figure 1: C program for Fibonacci. The right-hand-side comment is shown to easily identify recursive subcomputations with i .

Figure 2: Recursive subcomputations in $\text{fib}_1(51)$ in Fig. 1.

In many irregular applications, the *static* partitioning of work with sufficient concurrency into parallel tasks, 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 frameworks [2, 5, 8, 11, 13, 14, 20, 21] well implement load balancing.

As shown in Figures 1 and 2, let s_1 be a computation involving $\text{fib}_1(51)$, which contains two subcomputations (s_2) “ $r0 = \text{fib}_2(50)$ ” and (s_3) “ $r1 = \text{fib}_3(49)$ ”. Likewise, s_2 contains two (sub-)subcomputations (s_4) “ $r0 = \text{fib}_4(49)$ ” and (s_5) “ $r1 = \text{fib}_5(48)$ ”, and s_3 contains two (sub-)subcomputations (s_6) “ $r0 = \text{fib}_6(48)$ ” and (s_7) “ $r1 = \text{fib}_7(47)$ ”, and so on.

Let worker W_0 initially start s_1 and then worker W_2 steals s_3 from W_0 (with s_1). Subsequently, W_1 steals s_5 from W_0 (with $s_1 - s_3$) and W_3 steals s_7 from W_2 (with s_3). All four workers are successfully busy. When W_3 steals s_7 from W_2 , W_2 may *save* the stolen task s_7 as lightweight (local) checkpointing. When W_3 (with s_7) stops because of a fault, W_2 (with $s_3 - s_7$) can restart the saved stolen task s_7 . A work-stealing framework with this strategy is seemingly fault-tolerant. However, in case of a fault at W_2 , W_0 has to restart a large portion (s_3) of divide-and-conquer computation. In addition, without propagating and reusing the (partial) result of s_7 [22] or more precisely tracking and reconstructing the relationship [12], the restarted s_3 may be redundant and interfere with W_3 ’s work (s_7). Moreover, such a work-stealing framework is not fully fault-tolerant because W_0 is a single point of failure.

3 OUR APPROACH

This paper proposes the HOPE model for fault-tolerant, scalable, and efficient load balancing.

3.1 The Key Idea

Let us consider again the doubly recursive algorithm in Figures 1 and 2. For fault tolerance, every worker executes the same program;

for example, $\text{fib}_1(51)$ is executed. One may notice that the order of the computations is not highly significant at some points; for example, $\text{fib}_1(51)$ may execute $r0 = \text{fib}_2(50)$ and $r1 = \text{fib}_3(49)$ in this order, and also may execute in the other order of $r1 = \text{fib}_3(49)$ and $r0 = \text{fib}_2(50)$.

There are various orders that do not change the result of the program. We can employ multiple workers for the same program but in multiple orders in parallel. Such **SPMO** (single program, multiple order) parallel execution is fault-tolerant without any special handling of resilience, but such redundant SPMO execution itself does not provide any speedups. However, SPMO workers can exchange partial results to omit some computations and achieve parallel speedups.

We propose for every worker to use its own planned execution order at each recursive/hierarchical depth. For example, when using a notation similar to that in Section 2, we make W_0 , W_1 , W_2 , and W_3 perform s_1 as s_2s_3 , s_2s_3 , s_3s_2 , and s_3s_2 , respectively. During the execution of s_1 at depth 0 in Figure 2, W_0 and W_1 first execute s_2 (s_i with even i) as depth 1; we call this order *0-first* at depth 0. Similarly, W_2 and W_3 use *1-first* at depth 0.

At depth 2, we make W_0 , W_1 , W_2 , and W_3 perform s_1 as $s_4s_5s_6s_7$, $s_5s_4s_7s_6$, $s_6s_7s_4s_5$, and $s_7s_6s_5s_4$, respectively; that is, at depth 1, W_0 and W_2 use *0-first*, and W_1 and W_3 use *1-first*.

There are many runtime adaptive scenarios with these planned orders. If we could ignore more details at depth 3, one of many scenarios would be as follows: W_0 with $s_4s_5s_6s_7$ omits s_5 by receiving its result from W_1 , W_1 with $s_5s_4s_7s_6$ omits s_4 by receiving its result from W_0 , W_2 with $s_6s_7s_4s_5$ omits s_7 by receiving its result from W_3 , and W_3 with $s_7s_6s_5s_4$ omits s_6 by receiving its result from W_2 . W_0 , W_1 , W_2 , and W_3 compute the results of s_2 , s_2 , s_3 , and s_3 , respectively. W_0 with s_2s_3 omits s_3 by receiving its result from W_2 , W_1 with s_2s_3 omits s_3 by receiving its result from W_3 , W_2 with s_3s_2 omits s_2 by receiving its result from W_0 , and W_3 with s_3s_2 omits s_2 by receiving its result from W_1 . Finally, W_0 , W_1 , W_2 , and W_3 compute the same results (the value of $\text{fib}_1(51)$) of s_1 , s_1 , s_1 , and s_1 , respectively. Note that we do not need to wait for slow workers; we can use the result of the fastest worker.

We can make W_0 , W_1 , W_2 , and W_3 perform s_4 (at depth 2) as s_8s_9 (0-first), s_8s_9 (0-first), s_9s_8 (1-first), and s_9s_8 (1-first), respectively, or more sophisticatedly (as discussed in Section 6.5) perform s_4 as s_8s_9 (0-first), s_9s_8 (1-first), s_9s_8 (1-first), and s_8s_9 (0-first), respectively. If we consider at depth 3 (or more) or irregular grains of subcomputations, communication patterns are adaptively changed to omit more subcomputations.

3.2 General/Theoretical Definitions

Let us introduce *variable-length addresses* to more generally identify subcomputations for exchanging the partial results. When the entire work can be recursively divided into subcomputations, we can realize a tree of subcomputations as in Figure 3. We can identify each subcomputation with a variable-length address such as “@”, “@1”, “@01”, and “@12”. Note that variable-length addresses are similar to *pedigrees* in Cilk Plus [10]. With variable-length addresses, s_9 and s_{10} in Figure 2 are identified as @001 and @010, respectively. In general, we can use the following syntax for representing the structure of hierarchical “multiple-order” computation

Figure 3: Hierarchical “multiple-order” computation and variable-length addresses (pedigrees). The circle indicates a nondeterministic execution order for a pair of subcomputations. When “s” at the root address “@” contains a pair of subcomputations {s0, s1}, s0 is at @0 and s1 is at @1. When s0 contains subcomputations {s00, s01}, s01 is at variable-length address @01; when s1 contains two pairs of subcomputations {s10, s11} and {s12, s13}, s12 is at variable-length address @12.

(NOT programs):

subcomputations	$s ::= [c; c; \dots; c]$
components	$c ::= p \mid b$
pairs of subcomputations	$p ::= \{s, s\}$
basic components	$b ::= \dots$

For example, when computation s is $[r = \sum_{i=0}^{n-1} f(i)]$, a simple “multiple-order” version can be

$$[(m = n/2); \{[(r_0 = \sum_{i=0}^{m-1} f(i)), [(r_1 = \sum_{i=m}^{n-1} f(i))]; (r = r_0 + r_1)].$$

By rewriting $[(r_0 = \sum_{i=0}^{m-1} f(i))]$ recursively as much as possible, we can generate more versions.

When a worker uses the result of a subcomputation, the worker can omit the subcomputation. We can apply “work omission” hierarchically. When P workers run and each worker omits $(P-1)/P$ of the entire work, they achieve an ideal parallel speedup; moreover, when at least one of the P workers omits nearly $(P-1)/P$ of the entire work, it nearly achieves an ideal parallel speedup. Thus, it would like to receive at least hierarchical $O(\log P)$ results from other workers for the near-ideal parallel speedup.

Two major issues to be addressed in HOPE are as follows: (1) scalability and (2) overhead (efficiency). The following sections describe the design and implementation of the HOPE framework, including the language, compiler, and runtime system.

4 THE HOPE LANGUAGE

In the current design of the HOPE language, we use C as the base language and extend it with HOPE directives. HOPE directives can be used to (hierarchically) specify which regions in imperative code can be executed in sequential but arbitrary order and how their results can be accessed.

As in Figure 4, the `do_two` directive specifies that two omissible statements written immediately after it can be executed in arbitrary order. The omissible directive specifies that the execution

```

1  int fib(int n) {
2    if (n <= 2) return 1;
3    int r0, r1;
4    #pragma hope do_two // may execute in arbitrary order
5    #pragma hope omissible result(r0) // r0 is the partial result
6    r0 = fib(n - 1); // may exchange r0 for omission
7    #pragma hope omissible result(r1) // r1 is the partial result
8    r1 = fib(n - 2); // may exchange r1 for omission
9    return r0 + r1;
10 }

```

Figure 4: HOPE program for Fibonacci.

of the statement written immediately after it can be omitted if the result of the statement is available. The execution of the statement should be *idempotent*; that is, multiple-time execution should only produce the same result redundantly. The `result` clause specifies the location of the result (syntactically, an *lvalue* extended with array notation [10]). In Figure 4, `r0` and `r1` are specified. Note that the location of a result is not always the left-hand side of an assignment statement. For realistic applications, complex partial results such as sparse (sub)matrices, lists, trees, and graphs, are handled as general-purpose reader/writer (instead of `result`) clauses based on standard (de)serialization techniques.

Like `do_two` directive, HOPE for directive specifies that all iterations (of the omissible *body* statement) can be executed in arbitrary order. Some application programs in Section 8 use this directive. The HOPE compiler internally converts iterations into a hierarchical binary tree of subcomputations.

As idioms, both `do_two` and `for` directives must be applied to omissible statements.

Future versions may support directives other than `do_two` and `for`. (Currently, a manual combination is required.)

5 BASE IMPLEMENTATION

This section shows a base implementation, which will be improved in the following sections.

For simplicity, we employ an OS’s *process* for each worker. (In future version, we may employ a process for multiple workers (worker threads).)

Every worker employs an underlying *message mediation subsystem* (MMS). The MMS for a worker is located in the same process as the worker. As OS processes are distributed, MMSs are distributed and federated, each serving as a local storage and an automatic exchanger of *messages*. Each *message* contains the variable-length address and the result of a subcomputation. A message is a *fact* (universal information) to be idempotently discovered by any workers and exchanged by any workers and MMSs.

Because a set of variable-length addresses constitutes a tree structure, each MMS employs a *trie* for the efficient local storage of messages. A worker uses a *cursor* to specify the variable-length address of the *current* subcomputation for which the worker wants to read/write/check a partial result from/into/at the MMS. The MMS expands the trie as the cursor moves. The cursor also serves as a *reference* and the MMS performs local weighted reference counting for local storage reclamation. A trie node with zero reference counts can be deallocated if it does not contain child nodes and the result of one of its ancestor subcomputations is available. (We

call such ancestor subcomputations *super-subcomputations*.) As a cursor moves up or a new result is written, the MMS recursively deletes deallocatable descendants.

Each MMS employs a *communication and notification layer* (CNL) as a lower-level underlying layer. The CNL supports *system messages* between MMSs. Each MMS registers a handler of system messages. The CNL for an MMS is located in the same process as the MMS. CNLs are currently implemented with MPI and POSIX threads. Each CNL employs two threads each for handling external system messages and calling the registered system message handler. To avoid the case where some blocking is propagated into many MMSs, MMSs are loosely coupled by avoiding long-term blocking communications.

In each process, three threads exist (other than for the MPI implementation): a single worker thread and two CNL threads. Since the worker thread and a CNL thread may call MMS APIs and the MMS system message handler simultaneously, pthreads’s mutual exclusion is applied. We avoid deadlock by using mutual exclusion only with the partial order from the root to leaves. Furthermore, we avoid long-term blocking by providing only such MMS APIs.

When compiling a HOPE program, the HOPE compiler embeds calls to MMS APIs. At the beginning and end of each hierarchical subcomputation (such as s_2 , s_4 , s_8 , $\dots$, and $s_{2^{49}}$ in Figure 2), the worker (executing the compiled HOPE program) checks if the result of the subcomputation is locally available at its underlying MMS. If it is not available at the end, the result of the subcomputation is written into the MMS. If it is available, the result is used to omit the subcomputation.

Occasionally, only the result of a *super-subcomputation* is available. In such a case, the worker omits the subcomputation and recovers the correctness of the entire work using the correct result of the *super-subcomputation*.

When workers plan proper orders and perform subcomputations less redundantly and distributed MMSs properly exchange (propagate) *messages*, the base implementation embodies the concept of the HOPE model.

6 PRE-SHARED PLANS FOR SCALABILITY

For improving scalability, every worker executes hierarchical subcomputations based on its own planned order instead of randomly selected order. Note that each plan only determines the order; it does not *directly* determine the communication patterns.

The plan of each worker depends only on the rank of the worker and the total number of all workers. As each plan, we use a cyclic bit sequence; the d th bit determines the execution order of a pair (i.e., 0-first or 1-first) at depth d (See Figures 2 and 3). The cycle of the bit sequence is discussed in Section 6.5.

The plans of all workers are pre-shared (common and invariant) among workers; that is, the plan of any worker can be computed as a pre-determined function of the rank of the worker and the total number of all workers.

We arrange the entire pre-shared plans so that workers are divided into selectable hierarchical orders as evenly as possible. This scheme will provide good load balancing by making different workers start different portions of the entire work.

The MMS for a worker uses the pre-shared information and the current and varying local availability of messages to guess whether some other worker is currently participating in a hierarchical subcomputation. By carefully arranging the entire pre-shared plans, the MMS can quickly count and enumerate the estimated participants of the subcomputation.

6.1 Mostly Well-Matched One-to-One Correspondence

Using an estimated worker set, each MMS can send the result of a subcomputation only to the relevant MMSs. This makes the HOPE implementation more scalable. Furthermore, each MMS can perform mostly *well-matched* one-to-one correspondences.

Let us consider again the example in Section 3.1. For the result of s_3 (“ $r1 = fib_3(49)$ ”), all four combinations where either of W_0/W_1 receives a message from either of W_2/W_3 are possible for a one-to-one correspondence. If W_2 can select W_0 and W_3 can select W_1 independently, two (instead of four) combinations will suffice for covering the users of the result of s_3 . Such selections can be based on the estimated (last minute) participants of the hierarchically divided subcomputations: s_3 is divided into s_6 (W_2) and s_7 (W_3), and s_2 is divided into s_4 (W_0) and s_5 (W_1), where they can obtain the correspondence of W_2 to W_0 and that of W_3 to W_1 ,

6.2 Catch-up Information

MMSs use system messages for good collaboration. For example, catch-up information including multiple messages (partial results) is useful for a new participant of a subcomputation.

Let us consider again the example in Section 3.1. When W_0 receives a message for s_5 from W_1 , W_0 can guess that W_1 has become a new participant of s_4 . Thus, W_0 replies the catch-up information on s_4 to W_1 so that W_0 and W_1 can collaborate on the internal subcomputations (such as s_8 , s_9 , s_{16} , s_{17} , $\dots$, $s_{2^{49}}$, and $s_{2^{49}+1}$) within s_4 . Likewise, when W_0 receives a message for s_3 from W_2 , W_0 can guess that W_2 has become a new participant of s_2 . Thus, W_0 replies the catch-up information on s_2 to W_2 so that W_0 and W_2 can collaborate in internal subcomputations (such as s_4 , s_5 , s_8 , s_9 , $\dots$, $s_{2^{49}}$, and $s_{2^{49}+1}$) within s_2 .

As catch-up information on s_i , we use a chain of available results along with a subtree path from the subtree root s_i to one of its leaves. The one-to-one correspondence shown in Section 6.1 is well matched in terms of obtaining useful catch-up information.

6.3 Prefetch

MMSs can use system messages for prefetching. Immediately before the completion of a subcomputation, the MMS of a worker can request catch-up information on the subcomputation that the worker will soon take part in.

For this purpose, an MMS estimates the completion rate of a subcomputation based on the hierarchically divided subcomputations of the subcomputation and sends the completion rate in some steps. The receiver of the completion rates may interpret high completion rates as requests of catch-up information.

6.4 Suppressing Excess Collaboration

When most of the entire computation is completed, many workers would take part in a remaining finer-grained subcomputation s_{small} ; that is, the MMSs of the workers collaborate on s_{small} . However, excess collaboration among the MMSs is harmful because the workers can only omit s_{small} or hierarchical subcomputations of s_{small} at the costs of many system messages.

We propose to suppress excess collaboration; each MMS monitors the number n_{out} of its outstanding system messages that request catch-up information. When n_{out} exceeds a certain limit (e.g., 4 is used in Section 8), the MMS delays the requests until n_{out} is decreased. Delayed requests may be eliminated finally if the result of a *super*-subcomputation makes them wasteful. A reply or its timeout decreases n_{out} . Note that timeout makes this technique fault-tolerant.

6.5 Arranging the Entire Pre-Shared Plans

We propose to arrange the entire pre-shared plans (cyclic bit sequences) so that we can employ a fast algorithm for finding a worker set associated with a hierarchical subcomputation.

We consider a condition on association as a set of bit constraints. Starting with the entire worker set, our algorithm requires only constant space and time to represent the smaller set of associated workers by accepting a bit constraint.

When the total number of all workers is 16, we may use four-bit-cyclic bit sequences $b_0b_1b_2b_3-b_0b_1b_2b_3\cdots$ as *simple* pre-shared plans. Each combination of four bits corresponds to one of the 16 ($= 2^4$) workers; e.g., the plan of worker 0 is bit sequence 0000-0000- $\cdots$ and the plan of worker 3 is bit sequence 0011-0011- $\cdots$. In this situation, a given set of 0/1 constraints on some positions in the bit sequence (e.g., $\{b_0 = 0, b_7 = 1, b_{10} = 1\}$) can be “translated” into the constraints on four bits with modulo 4. For example, constraint $b_0 = 0$ only excludes $1b_1b_2b_3-1b_1b_2b_3\cdots$; that is, constraint $b_0 = 0$ does not exclude workers 0 and 3. Constraint $b_7 = 1$, which is translated into constraint $b_3 = 1$, excludes $b_0b_1b_20-b_0b_1b_20\cdots$; that is, constraint $b_7 = 1$ excludes worker 0.

With the *simple* pre-shared plans above, since $b_4 = 1$ is translated into constraint $b_0 = 1$, two constraints $b_0 = 0$ and $b_4 = 1$ exclude all workers; that is, there are no 1-first workers at depth 4 among eight 0-first workers at depth 0. To alleviate this “load Imbalance,” as a more sophisticated technique, we use an exclusive OR ($\oplus$) of two or more bits out of the base four bits. By adding such 11 combined bits b_4 to b_{14} (e.g., $b_4 = b_0 \oplus b_1 \oplus b_2 \oplus b_3$ and $b_{14} = b_0 \oplus b_3$) like Hamming(15, 11) codes [6], we can extend the four-bit cyclic to the 15-bit cyclic. Since constraint $b_4 = 1$ (or $b_4 = 0$) is translated into $b_0 \oplus b_1 \oplus b_2 \oplus b_3 = 1$ (or $b_0 \oplus b_1 \oplus b_2 \oplus b_3 = 0$), there are four 1-first workers and four 0-first workers evenly at depth 4 among eight 0-first workers at depth 0.

In the current implementation, we use all combinations of two or more bits out of k base bits if $0 \leq k \leq 4$. We use only additional $3k - 1$ combinations of two or more bits out of k base bits if $k > 4$. When the total number (n_w) of all workers is not a power of two, we select n_w sequences out of 2^k sequences with k such that $n_w \leq 2^k$.

7 REDUCING WORKER OVERHEADS

7.1 Adding the Non-check Mode

When compiling a HOPE program, the HOPE compiler embeds calls to MMS APIs. Each worker omits a subcomputation if the result of the subcomputation is locally available. Furthermore, it locally writes the result of the subcomputation into the MMS for other workers. It uses a *cursor* to specify the variable-length address (ID) of the *current* subcomputation for which it may locally read/write/check a partial result from/into/at the underlying MMS. As a cursor moves, the MMS expands and reduces its own *trie*.

These operations require significant maintenance/check overheads. For example, expanding/reducing a trie includes significant overheads with *malloc/free*; thus, we propose adding the “non-check” execution mode in an application-independent manner.

In the non-check mode, unlike the “check” execution mode above, each worker does not perform reading/writing/checking of messages or move the cursor (i.e., the MMS does not expand and reduce its trie). In the non-check mode, each worker still performs order selection operations based on its own plan.

We propose using the “non-check” mode when the worker is presumed to be the sole participant of the current subcomputation s_{cur} , as the result of s_{cur} is less likely to be locally available *and* less likely to be useful for other workers. Once the worker checks that the number of the estimated participants of some *super*-subcomputation⁴ of s_{cur} is only one (i.e., the worker is presumed to be sufficiently “alone”), it starts new *sub*-subcomputations in the non-check mode. It resumes s_{cur} in the check mode after the end of every *sub*-subcomputation.

Since this technique allows a lot of subtrees of a divide-and-conquer computation to be executed in the non-check mode, the average overheads are significantly reduced.

Additionally, we propose using the “non-check” mode for suppressing excess collaboration. The suppression proposed in Section 6.4 changes MMSs’ behaviors. The newly proposed suppression makes workers with suppressed MMSs run in the non-check mode.

7.2 Adding the Work-first Mode

The HOPE compiler embeds order selection operations based on pre-shared plans.

Even in the non-check mode, each worker incurs considerable overheads of order selection operations and branch prediction misses; thus, we propose adding the “work-first” execution mode, where the worker ignores its plan and simply uses the *default* order (i.e., that of HOPE-directive-free code).

We propose using the “work-first” mode when the worker is expected to remain the sole participant of the current subcomputation s_{cur} during the execution of s_{cur} . Currently, a worker starts new subcomputations in the “work-first” execution mode immediately after decuply nested non-check-mode subcomputations, that is, immediately after following the planned order selection (in non-check mode) 10 times in hierarchically nested subcomputations.

The HOPE compiler generates a separated clone of code for the “work-first” mode. Note that two clones (a fast clone and a slow clone) are already used in Cilk-5 [5] as a similar technique.

⁴We employ the grandparent-subcomputation of s_{cur} .

7.3 Dynamic Execution Mode Switching

In addition to the check, non-check, and work-first execution modes, we propose using a dynamic execution mode-switching technique.

If participants of a subcomputation increase later, the past choice of the “non-check” mode for the subcomputation is changed as if it were started in the “check” mode. For the reborn check-mode subcomputation, the worker moves the cursor to the right position and calls the MMS APIs for reading/writing/checking of messages. For writing a message, the worker reads the result deeply in the stack.

When a non-check-mode subcomputation dynamically becomes a check-mode subcomputation, the condition where a work-first-mode subcomputation follows decuply nested non-check-mode subcomputations becomes violated. To satisfy the condition, the past choice of the “work-first” mode for the subcomputation is changed later as if it were started in the “non-check” mode.

To access results and execution modes and other useful data deeply in the stack, we use nested functions as extended C languages’ mechanisms for legitimate execution stack access, such as GCC’s trampolines [1], closures [24] and L-closures [9, 24]. A similar dynamic execution mode-switching technique using mechanisms for legitimate execution stack access is already used in Tascell [8].

8 EVALUATION

The evaluation environments are summarized in Table 1. For compiling nested C functions, we employed the Intel Compiler on Xeon Phi and a translator into standard C [9] on the (Riken R-CCS) K computer.

We compare HOPE and Tascell using one or multiple nodes on the K computer primarily for evaluating *fault tolerance* and *scalability*. We compare HOPE, Tascell, and Cilk Plus [10] using one node (as a shared memory environment) on Xeon Phi primarily for evaluating *efficiency*. Note that Cilk Plus can run only in a shared memory environment.

In *both* environments, HOPE uses as many worker processes as workers; every interaction between workers is based on the MPI (i.e., flat MPI). Unless otherwise stated, we use the improved HOPE implementation with all three execution modes and dynamic execution mode switching (DEMS).

In *both* environments, Tascell uses a single process for each node that involves multiple workers (POSIX threads); fast intra-node interactions between workers are possible based on shared memory. For inter-node communication, a “Tascell server”-less implementation [15]⁵ is employed

For each configuration, we employ the median of five runs. Because any HOPE process can complete the entire computation and call `MPI_Abort`, the minimum of the available elapsed times is used. To measure elapsed times as precise as possible in HOPE, we employ `MPI_Barrier` at the start times.

In our evaluation, *efficiency* is defined as S/n_w , where S is a speedup relative to a *sequential C program* and n_w is the total number of workers (including workers with faults). (Efficiency = 1

Table 1: Evaluation environments

	Xeon Phi	The K computer (in 1 node)
Host processor	Intel Xeon E5-2697 v2 12-core × 2	SPARC64 VIIIx 2GHz 8-core
Coprocessor	Intel Xeon Phi 3120P 57-core × 2	-
Memory	64GB	16GB
Coprocessor Memory	6GB for each coprocessor	-
Network	Each coprocessor is connected to the host via PCIe 3.0×16 (Bandwidth = 15.6 GB/s)	Tofu Interconnect (Bandwidth = 5 GB/s per link)
OS	CentOS 6.5 (64bit)	-
Compiler	Intel Compiler 13.1.3 with -O2 optimizers	Fujitsu C/C++ Compiler 1.2.0 with -O2 optimizers
Nested Function	Trampoline-based implementation (compatible with the GCC extension [1])	LW-SC [9]
MPI library	Intel MPI 4.1 Update 3 Build 20140124	-

means an ideal speedup.) Notice that S is *not* relative to one-worker execution.

We evaluated the performance using the following programs:

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

Nq(n) finds all solutions to the n -queens problem on the basis of backtrack search.

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.

AreaSum(n) approximately computes the area of a quadrant by recursively dividing a square containing it into rectangles for summing up their contributions until rectangles are not across the boundary or their areas are less than $(10.0)^{-n}$.

MatMul(n) computes the matrix multiplication of an $n \times n$ matrix using a cache-oblivious recursive algorithm.

In Nq(n) and Pen(n), each HOPE worker can perform sequential backtrack search by *modifying* a single workspace without spawning tasks and copying the workspace. Unlike typical multi-threaded languages, Tascell [15] also supports on-demand concurrency where task spawning and workspace copying are delayed until work stealing occurs.

The search trees of Pen(n) and AreaSum(n) are similar to the unbalanced tree search [17] in the sense that most nodes do not generate large subtrees.

Fib(n), Nq(n), Pen(n), and AreaSum(n) do not use any *thresholds* (or *cut-offs*) to switch to *plain* sequential execution; for example, $r0 = \text{fib}(2)$ and $r1 = \text{fib}(1)$ are the finest grained subcomputations (in HOPE), potential tasks (in Tascell), or “logical” threads (in Cilk Plus).

MatMul(n) uses a *threshold* where submatrices less than 10×10 are computed ($\sim 10^2 n$ multiplication operations) by a *plain* triple nested for loop and saved into a shared memory object on each node. In Tascell, the object is located in shared memory. In HOPE, it is shared among intra-node worker processes with `shm_open`. In both languages, the elapsed times do not include inter-node collective communication for an $n \times n$ matrix.

⁵We enhanced the MPI-based implementation in [15] with multiple issues of `MPI_Isend`.

Figure 5: Efficiency (upper half) and execution times (lower half) of parallel systems using multiple workers on the K computer, in addition to execution times of serial C programs (lower half). We use 1 to 1024 nodes, each of which has eight cores and employs seven workers.

8.1 Fault Tolerance

We developed HOPE implementations and ensured their fault tolerance. However, our evaluation environments only provide MPI implementations in which the entire parallel execution is terminated if any MPI process exits. Thus, we employ fault injections by which one out of n MPI processes (workers) simply lose their received/sending messages.

Figure 5 shows the results of 1, 2, 4, ..., and 1024 nodes each with seven workers on the K computer. “HOPE FT(n)” stands for “HOPE with Fault injection into one out of n workers”; we artificially create an environment with total n_w workers where $\lfloor n_w/n \rfloor$ workers cannot communicate with other $n_w - 1$ workers.

It is evident that the performance difference between the configurations with and without faults on one out of 64 workers is small. In addition, even when only three out of four workers continue with communication, the entire computation is completed by some workers and reasonable efficiency is demonstrated.

8.2 Scalability

Figure 5 compares HOPE and Tascell using 1, 2, 4, ..., and 1024 nodes.

In terms of scalability and load balancing, as the number of workers increases, efficiency decreases. Notice that decreased elapsed

times approach to one second (i.e., challenging) with strong scaling. In most cases, HOPE exhibits scalability similar to Tascell, but exhibits slightly better scalability for MatMul(10000). Since MatMul(10000) uses a threshold, a Tascell thief may wait for a new task for a long time, whereas HOPE workers never wait for unavailable results.

8.3 Efficiency

Figure 6 shows the results on Xeon Phi. Both Tascell and HOPE demonstrate better performances than Cilk Plus except MatMul(10000). Tascell [8] achieves a low parallelization overhead, whereas HOPE programs Fib(n), AreaSum(n), and MatMul(n) exhibit similar parallelization overhead even though they support fault tolerance. HOPE programs Nq(n) and Pen(n) exhibit lower efficiency than Tascell. This is because the HOPE compiler *eagerly* converts (divides) iterations into a binary tree of subcomputations, whereas Tascell performs iterations which can be *lazily* divided on work stealing.

In Figure 6, the elapsed times of HOPE are not reduced when 128 workers are employed. This is because each node contains only 57 cores (228 hardware threads) and the current HOPE implementation requires two busy threads⁶ for each process.

⁶The use of a common thread will be addressed in future work, which may solve this problem.

Figure 6: Efficiency (upper half) and execution times (lower half) of parallel systems using multiple workers on Xeon Phi (using one node), in addition to execution times of serial C programs (lower half). HOPE uses as many processes as workers. Each of Tascell and Cilk™ Plus uses a single process with multiple worker threads.

Table 2: Execution times with and without three execution modes and dynamic execution mode switching (DEMS) on Xeon Phi with 16 workers (and execution times relative to the *all-on* implementation).

Implementations	execution modes				elapsed times [s] (and relative execution times)			
	check	non-check	work-first	DEMS	Fib(41)	Fib(51)	Pen(13)	Pen(15)
base + Sec. 6	on	off	off	off	433.3 (360.3)	-	907.8 (220.9)	-
w/ non-check	on	on	off	on	3.432 (2.853)	372.9 (3.256)	12.00 (2.919)	959.9 (3.110)
w/ work-first	on	on	on	on	1.203 (1.000)	114.5 (1.000)	4.109 (1.000)	308.7 (1.000)
w/o DEMS	on	on	on	off	1.228 (1.021)	122.5 (1.070)	13.01 (3.165)	432.2 (1.400)

8.4 Measuring Worker Overheads

Table 2 shows the execution times on Xeon Phi with and without the two additional execution modes and DEMS proposed in Section 7. The effectiveness of the non-check mode and work-first mode is evident. The execution times of the base implementation only with improvements from Section 6 are more than 220 times longer than the *all-on* implementation. The non-check mode effectively reduces the significant maintenance/check overheads (primarily for expanding/reducing a trie including malloc/free in the underlying MMS) of the check mode. The work-first mode effectively reduces the considerable overheads of order selection operations and branch prediction misses.

Table 2 shows the effectiveness of DEMS; some execution times without DEMS are more than 1.4 times longer than those with DEMS. Figure 7 also shows the effectiveness; without DEMS, some execution times with many workers on the K computer are more than twice longer than those with DEMS.

9 RELATED WORK

9.1 Exact and Inexact Work Division

In the HOPE framework, workers can redundantly assign a value to a common location as assignments are complete and idempotent. By contrast, workers cannot redundantly apply an effect to a common location if the effect is not complete or idempotent. For example, adding a value to a common location is neither complete nor idempotent.

Idempotent work stealing [13] is an example that permits inexact work division, if the execution of the same work by two workers is identical to the execution by one worker.

9.2 Fault Tolerance and Early Termination

One may assume that checkpointing can be used for work-stealing frameworks. We can use non-volatile storage for fault tolerance. However, without high-level semantics, the restarted thief cannot reuse the saved state in the global context.

Figure 7: Execution times with and without DEMS on the K computer. Each of 1 to 1024 nodes employs 7 workers.

X10 employs the asynchronous partitioned global address space (APGAS) programming model. Resilient X10 [3] is an extended X10 language and implementation that enables applications to recover from place failures, which may cause the loss of data and tasks at the failed place. Failure is exposed through exceptions. The programmer is responsible for continuing execution with fewer places, but the runtime automatically repairs the global control structure of the program to reduce much of the burden of resilient programming. In addition, Resilient X10 over MPI User Level Failure Mitigation has been developed [7]. In HOPE, control is inherently redundant; the HOPE programmer is not required to write code for resilience except some directives. In our opinion, HOPE over MPI User Level Failure Mitigation will be a promising and straightforward direction.

Unlike failure-oblivious computing [18] (although [18] is studied in the context of server availability and security), the HOPE framework does not use incorrect or approximate data, and can survive even when only one worker remains.

Rinard proposed early phase termination to eliminate load imbalances at barrier synchronization points [19]. This approach also uses approximation. The HOPE framework does not use approximation, but most workers can be terminated earlier than some workers as only the best-effort exchanging of partial results exist without synchronization.

9.3 Redundant Parallel Execution

Several forms of redundant parallel execution exist. These approaches employ mechanisms for redundancy elimination.

Some parallel SAT solvers [16] employ the *lemma exchange*. Parallel (game) tree search often employs transposition tables. SPMD (e.g., XcalableMP [23]) workers perform communication.

10 CONCLUSION

This paper proposed a new “hierarchical omission”-based parallel execution model called HOPE. There are various orders that do

not change the result of the program. We can employ multiple workers for the same program but in multiple orders in parallel. Such SPMD parallel execution itself does not provide any speedups. However, SPMD workers can exchange partial results for omitting some computations and achieve parallel speedups.

The HOPE programming/execution framework provides efficient and fault-tolerant dynamic load balancing. The design and implementation of the HOPE framework were described herein, including the language, compiler, and runtime system.

To reduce checking costs and omit redundant computations simultaneously, we added the *non-check* mode and *work-first* mode other than the check mode and employed nested functions to apply the check mode only to subcomputations that were not sufficiently alone with respect to other workers. Furthermore, the *work-first* mode avoids order selection operations and branch prediction misses as much as possible.

For scalability, we associated each subcomputation with a set of workers, and propagated a partial result only to relevant workers only when it was useful. We used a fast algorithm for finding a worker set associated with a hierarchical subcomputation.

ACKNOWLEDGMENTS

This work was supported in part by JSPS KAKENHI Grant Numbers JP19H04087 and JP17K00099. Part of the results are obtained by using the K computer at the RIKEN Center for Computational Science. This work was conducted in part in collaboration with RIKEN R-CCS for “Research and development to improve the programming environment of the K computer.”

REFERENCES

- [1] Thomas M. Breuel. 1988. Lexical Closures for C++. In *Usenix Proceedings, C++ Conference*.
- [2] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.* 40, 10 (2005), 519–538. <https://doi.org/10.1145/1103845.1094852>
- [3] David Cunningham, David Grove, Benjamin Herta, Arun Iyengar, Kiyokuni Kawachiya, Hiroki Murata, Vijay Saraswat, Mikio Takeuchi, and Olivier Tardieu. 2014. Resilient X10: Efficient Failure-aware Programming. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*. 67–80.
- [4] Graham E. Fagg and Jack Dongarra. 2000. FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer-Verlag, London, UK, 346–353.
- [5] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. *ACM SIGPLAN Notices (PLDI '98)* 33, 5 (1998), 212–223.
- [6] R. W. Hamming. 1950. Error Detecting and Error Correcting Codes. *Bell System Technical Journal* 29, 2 (1950), 147–160.
- [7] Sara S. Hamouda, Benjamin Herta, Josh Milthorpe, David Grove, and Olivier Tardieu. 2016. Resilient X10 over MPI User Level Failure Mitigation. In *Proceedings of the 6th ACM SIGPLAN Workshop on X10 (X10 2016)*. 18–23.
- [8] Tasuku Hiraishi, Masahiro Yasugi, Seiji Umatani, and Taiichi Yuasa. 2009. Backtracking-based Load Balancing. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2009)*. 55–64.
- [9] Tasuku Hiraishi, Masahiro Yasugi, and Taiichi Yuasa. 2006. A Transformation-Based Implementation of Lightweight Nested Functions. *IPSJ Digital Courier* 2 (2006), 262–279. (IPSJ Transaction on Programming, Vol. 47, No. SIG 6(PRO 29), pp. 50–67.).
- [10] Intel Corporation. [n. d.]. Introducing Intel® Cilk™ Plus: Extensions to simplify task and data parallelism. <http://www.cilkplus.org>.
- [11] Intel Corporation. 2007. *Intel Threading Building Block Tutorial*. <http://threadingbuildingblocks.org/>.
- [12] Gokcen Kestor, Sriram Krishnamoorthy, and Wenjing Ma. 2017. Localized Fault Recovery for Nested Fork-Join Programs. In *Proceedings of 31st IEEE International*

- Parallel and Distributed Processing Symposium (IPDPS 2017)*. 397–408. <https://doi.org/10.1109/IPDPS.2017.75>
- [13] Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. 2009. Idempotent Work Stealing. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2009)*. 45–54.
 - [14] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. 1991. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems* 2, 3 (July 1991), 264–280.
 - [15] Daisuke Muraoka, Masahiro Yasugi, Tasuku Hiraishi, and Seiji Umatani. 2016. Evaluation of an MPI-Based Implementation of the Tascell Task-Parallel Language on Massively Parallel Systems. In *Proceedings of the 45th International Conference on Parallel Processing Workshops (ICPPW 2016) (Ninth International Workshop on Parallel Programming Models and Systems Software for High-End Computing P2S2 2016, held in conjunction with ICPP 2016)*. 161–170. <https://doi.org/10.1109/ICPPW.2016.36>
 - [16] Kei Ohmura and Kazunori Ueda. 2009. c-sat: A Parallel SAT Solver for Clusters. In *Theory and Applications of Satisfiability Testing - SAT 2009*, Oliver Kullmann (Ed.). Lecture Notes in Computer Science, Vol. 5584. Springer Berlin Heidelberg, 524–537. https://doi.org/10.1007/978-3-642-02777-2_47
 - [17] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. 2007. UTS: An Unbalanced Tree Search Benchmark. In *Proceedings of the 19th International Conference on Languages and Compilers for Parallel Computing (LCP'06)*. 235–250.
 - [18] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebe, Jr. 2004. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*. 21–21.
 - [19] Martin C. Rinard. 2007. Using Early Phase Termination to Eliminate Load Imbalances at Barrier Synchronization Points. In *Proceedings of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2007)*. 369–386.
 - [20] Kenjiro Taura, Kunio Tabata, and Akinori Yonezawa. 1999. StackThreads/MP: Integrating Futures into Calling Standards. In *Proceedings of ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP'99)*. 60–71.
 - [21] Seiji Umatani, Masahiro Yasugi, Tsuneyasu Komiya, and Taiichi Yuasa. 2003. Pursuing Laziness for Efficient Implementation of Modern Multithreaded Languages. In *Proceedings of the Fifth International Symposium on High Performance Computing (Lecture Notes in Computer Science)*. 174–188.
 - [22] Gosia Wrzesińska, Rob V. van Nieuwpoort, Jason Maassen, Thilo Kielmann, and Henri E. Bal. 2006. Fault-Tolerant Scheduling of Fine-Grained Tasks in Grid Environments. *The International Journal of High Performance Computing Applications* 20, 1 (2006), 103–114. <https://doi.org/10.1177/1094342006062528>
 - [23] XcalableMP Specification Working Group. 2018. XcalableMP Language Specification Version 1.4.
 - [24] Masahiro Yasugi, Tasuku Hiraishi, and Taiichi Yuasa. 2006. Lightweight Lexical Closures for Legitimate Execution Stack Access. In *Proceedings of the Fifteenth International Conference on Compiler Construction (CC2006) (Lecture Notes in Computer Science)*. Springer-Verlag, 170–184.